Annexe 2 : Code

Plan de l'annexe 2

In [2]:

```
### Importations

# Pour pouvoir utiliser les tableaux
import numpy

# Pour générer un nombre aléatoire
import random

# Pour lire un fichier csv
import csv
```

1 Analyse de la base de données

1.1 Récupérer la base de données

La base de données choisie contient 20 000 263 ratings.

In [62]:

```
### Récupérer le contenu du fichier et le stocker dans la variable f
fichier = open ('ratings_ 20 000 000.csv')
f = csv.reader(fichier)
```

In [63]:

```
### Récupérer l'ensemble des éléments

# Variable qui correspond à la ligne que l'on parcourt
# -1 car la première ligne est celle des titres des colonnes
i=-1

# Variable qui contiendra une liste où chaque élément correspond à une ligne du fichier
liste_ratings = []

# Parcourir toutes les lignes du fichier
for ligne in f:

    # Ne pas lire la première ligne qui contient les titres des colonnes
    if(i!=-1):

        # Créer une liste qui contiendra l'ensemble des éléments de la ligne
        liste_ratings.append([])

        # Ajouter tous les éléments de la ligne
        liste_ratings[i].append(int(ligne[0]))
        liste_ratings[i].append(int(ligne[1]))
        liste_ratings[i].append(float(ligne[2]))
        liste_ratings[i].append(int(ligne[3]))

    # Passer à la ligne suivante
    i=i+1

# Convertir la liste en tableau
BD = numpy.asarray(liste_ratings)
print("Base de données récoltée :")
print(BD)

print("Il y a " + str(len(BD)) + " ratings.")
```

```
Base de données récoltée :
[[1.00000000e+00 2.00000000e+00 3.50000000e+00 1.11248603e+09]
 [1.00000000e+00 2.90000000e+01 3.50000000e+00 1.11248468e+09]
 [1.00000000e+00 3.20000000e+01 3.50000000e+00 1.11248482e+09]
 ...
 [1.38493000e+05 6.96440000e+04 3.00000000e+00 1.26020946e+09]
[1.38493000e+05 7.02860000e+04 5.00000000e+00 1.25812694e+09]
[1.38493000e+05 7.16190000e+04 2.50000000e+00 1.25581114e+09]]
Il y a 20000263 ratings.
```

Par la suite, nous aurons besoin de diviser la base de données en 5, 10 et 20. Or, elle n'est actuellement pas divisible par ces nombres. Donc, on enlève 3 lignes au hasard pour alors avoir une matrice de 20 000 260 ratings.

In [33]:

```
### Enlever 3 lignes au hasard
for i in range(3):
    BD = numpy.delete(BD, (random.randint(0,len(BD)-1)), axis=0)
```

1.2 Ordonner la base de données en fonction du temps

In [34]:

```
### Ordonner la base de données en fonction du timestamp, càd le 4ème élément
BD = BD[BD[:,3].argsort()]
```

In [35]:

```
print("Taille de la base de données : ",len(BD))
```

```
Taille de la base de données :  20000260
```

1.3 Diviser la base de données en *x* parties

In [36]:

```
### Méthode qui divise la base de données en "x" parties

def Division_BD (DataBase,x):
    divisions=numpy.split(DataBase,x)
    return divisions
```

In [38]:

```
### Diviser la base de données en plusieurs parties

# 5 parties
Division_BD_5 = Division_BD(BD,5)

# 10 parties
Division_BD_10 = Division_BD(BD,10)

# 20 parties
Division_BD_20 = Division_BD(BD,20)
```

1.4 Vérifier que les utilisateurs soient dans chacune des parties

1.4.1 Obtenir la liste des utilisateurs présents dans toutes les partie

Pour rappel, nous souhaitons conserver uniquement les utilisateurs présents dans toutes les divisions de la base de données afin de pouvoir faire une analyse temporelle des recommandations qui leur sont faites. Dès lors, nous avons tout d'abord créé la méthode *Utilisateurs* qui permet de fournir une liste des identifiants de tous les utilisateurs présents dans la base de données fournie en paramètre. Ensuite, nous trouvons la liste des utilisateurs présents à la fois dans chacune des parties.

Nous obtenons alors les résultats suivants :

- Pour la base de données de 100 000 ratings, nous trouvons 23 utilisateurs lorsque la base de données est divisée en 5 parties et seulement 2 utilisateurs lorsqu'elle est divisée en 10.
- Pour la base de données d'1 000 000 de ratings, nous trouvons 24 utilisateurs lorsque la base de données est divisée en 5 parties.
- Pour la base de données de 10 000 000 de ratings, nous trouvons 181 utilisateurs lorsque la base de données est divisée en 5 parties et nous ne trouvons aucun utilisateur commun lorsqu'elle est divisée en 10.
- Pour la base de données de 20 000 000 de ratings, nous trouvons 335 utilisateurs lorsque la base de données est divisée en 5 parties et nous ne trouvons aucun utilisateur commun lorsqu'elle est divisée en 10.

Suite à cela, nous choisissons d'utiliser la base de données de 20 000 000 ratings et de diviser celle-ci en 5 parties afin d'avoir suffisamment de données pour réaliser notre étude.

In [39]:

```
### Méthode déterminant la liste des utilisateurs présents dans une base de données

def Utilisateurs (DataBase):

    # Liste qui contiendra l'ensemble des utilisateurs repris dans cette base de données
    liste_user = []

    # Parcourir toutes les lignes
    for i in DataBase:

        # L'identifiant de l'utilisateur est le premier élément de la ligne
        user = i[0]

        # Si l'utilisateur n'est pas encore présent dans la liste, l'ajouter à celle-ci
        if user not in liste_user:
            liste_user.append(user)

    return liste_user
```

In [44]:

```
### Trouver la liste des utilisateurs présents dans toutes les divisions

# Division utilisée
Division_BD = Division_BD_5
Partie1_BD = Division_BD[0]

# L'ensemble des utilisateur présents dans cette première partie
users_part1 = Utilisateurs(Partie1_BD)

# Liste reprenant les utilisateurs apparaissant dans toutes les parties,
# initialisée avec les utilisateurs de la 1ère partie
liste_users = users_part1

# Vérifier que tous ces utilisateurs sont bien présents dans les autres parties
for i in range(len(Division_BD)-1):
    users_partie_suivante = Utilisateurs(Division_BD[i+1])
    liste_users = list(set(liste_users).intersection(users_partie_suivante))

# Imprimer les résultats
print("Liste des utilisateurs présents dans toutes les parties :")
print(liste_users)
print("Nombre total d'utilisateurs :",len(liste_users))
```

```
Liste des utilisateurs présents dans toutes les parties :
[]
Nombre total d'utilisateurs : 0
```

1.4.2 Mettre à jour les bases de données en conservant uniquement les utilisateurs sélectionnés

Nous créons la méthode *Delete_Users* qui permet de renvoyer une nouvelle base de données contenant uniquement les informations relatives aux utilisateurs présents dans toutes les divisions.

In [45]:

```
### Méthode qui permet de supprimer les lignes
# référant à des utilisateurs qui n'appartiennent pas à la liste

def Delete_Users (DataBase, users_list):

    # Base de données qui contiendra les informations relatives
    # uniquement aux utilisateurs de la liste
    NewDataBase = []

    for i in DataBase:

        # L'id de l'utilisateur correspond au premier élément de la ligne
        user = i[0]

        # Vérifier si l'utilisateur est dans la liste
        if user in users_list:
            NewDataBase.append(i)

    # Renvoyer la nouvelle base de données
    return NewDataBase
```

In [46]:

```
### Corriger les bases de données pour conserver uniquement les utilisateurs de la liste

# Ensemble des divisions de la base de données
# mais en conservant uniquement les utilisateurs de la liste
Division_BD_corrigee = []

for division in Division_BD:
    Division_BD_corrigee.append(Delete_Users(division, liste_users))
```

1.5 Ecrire dans des fichiers CSV les différentes parties de la base de données

Nous écrivons chacune des divisions de la base de données corrigée dans un fichier CSV différent afin de ne pas devoir réexécuter tout le travail sur la base de données initiale.

In [ ]:

```
### Ecrire dans un fichier CSV les différentes parties de la base de données

# Variable qui permet de savoir de quelle division il s'agit
num_div = 1

# Parcourir l'ensemble des divisions de la base de données
for division in Division_BD_corrigee:

    # Ecrire dans un fichier CSV
    with open("Base de données 20 millions divisée en 5 - Partie " + str(num_div) + ".csv",

        # Variable pour écrire dans le fichier CSV
        division_csv = csv.writer(file,delimiter=';',lineterminator='\n')

        # Parcourir toutes les lignes pour écrire chacune d'elles
        for ligne in division:
             division_csv.writerow(ligne)

    # Passer à la division suivante
    num_div = num_div + 1
```

1.6 Récupérer les informations nécessaires des fichiers CSV

Il n'est pas nécessaire d'exécuter les points 1.1. à 1.5. (pour autant que ceux-ci aient déjà été exécutés au moins une fois auparavant ou que nous disposions déjà des fichiers CSV contenant les données de chaque division temporelle). nous allons récolter les informations nécessaires qui avaient été enregistrées dans les fichiers CSV précédemment.

1.6.1 Avoir une liste de tous les utilisateurs et une de tous les films

Tout d'abord, nous avons créé une méthode qui permet de récupérer les identifiants de l'ensemble des utilisateurs considérés dans la nouvelle base de données ainsi que l'ensemble des films qui ont été évalués par ces utilisateurs. Pour cela, nous parcourons les 5 divisions de la base de données reprises chacune dans un fichier CSV. Nous ajoutons alors tous les utilisateurs à la liste "tous_les_users" et tous les films à la liste "tous_les_films". Cette méthode renvoie ces deux listes.

In [3]:

```
### Méthode qui renvoie une liste reprenant les id de tous les utilisateurs
# et une reprenant les id de tous les films

def Lists_Users_And_Movies ():

    # Liste de tous les utilisateurs
    tous_les_users = []

    # Liste de tous les films
    tous_les_films = []

    # Lire l'ensemble des fichiers
    for div in range(1,6):

        # Lecture du fichier
        fichier = open ("Base de données 20 millions divisée en 5 - Partie " + str(div) + "
        f = csv.reader(fichier, delimiter=';')

        # Parcourir des lignes du fichier une à une
        for ligne in f:

            # Récupérer l'id de l'utilisateur et l'id du film
            user_id = int(float(ligne[0]))
            movie_id = int(float(ligne[1]))

            # Si l'id de l'utilisateur n'est pas dans la liste, on l'ajoute
            if user_id not in tous_les_users:
                tous_les_users.append(user_id)

            # Si l'id du film n'est pas dans la liste, on l'ajoute
            if movie_id not in tous_les_films:
                tous_les_films.append(movie_id)

    # Renvoyer les deux listes
    return tous_les_users,tous_les_films
```

In [4]:

```
### Récupérer les deux listes et imprimer le nombre d'utilisateurs et le nombre de films

tous_les_users, tous_les_films = Lists_Users_And_Movies()
print("Nombre d'utilisateurs : " + str(len(tous_les_users)))
print("Nombre de films : " + str(len(tous_les_films)))
```

```
Nombre d'utilisateurs : 335
Nombre de films : 15738
```

1.6.2 Réaliser les matrices binaires à partir des fichiers CSV

Nous avons réalisé une méthode permettant de récupérer le contenu d'un fichier CSV (enregistré précédemment) et de l'organiser sous forme de matrice binaire où les lignes correspondent aux utilisateurs, les colonnes aux films. Le contenu de la matrice vaut 0 si l'utilisateur n'a pas vu le film et 1 si l'utilisateur a vu le film. Les timestamps ne nous intéressent donc plus étant donné que la base de données initiale a déjà été séparée en plusieurs parties en fonction du timestamp. Pour savoir à quel utilisateur une certaine ligne fait référence, c'est-à-dire connaître l'identifiant de cet utilisateur, nous disposons de la liste "tous_les_users" créée précédemment où la position de l'identifiant d'un utilisateur dans la liste correspond à la position de la ligne dans la matrice reprenant les ratings. Le même principe est appliqué pour les films : la position de l'identifiant d'un film dans la liste de "tous_les_films" correspond à l'indice de la colonne dans la matrice de ratings.

Cette méthode renvoie alors la matrice "matrix_ratings" reprenant 0 (lorsque l'utilisateur n'a pas vu le film) et 1 (lorsque l'utilisateur a vu le film) uniquement pour la période de temps considérée.

In [5]:

```
### Méthode qui renvoie une matrice binaire (lignes = utilisateurs, colonnes = films)
### dont le contenu vaut 1 si l'utilisateur a vu le film, 0 sinon

def Matrix_Ratings (list_users, list_movies, filename):

    # Créer la matrice de cette période de temps en y mettant uniquement des zéros
    matrix_ratings = numpy.zeros(shape=(len(list_users),len(list_movies)))

    # Lire le fichier CSV
    fichier = open (filename)
    f = csv.reader(fichier, delimiter=';')

    # Récupérer les informations ligne par ligne
    for ligne in f:

        # Récupérer l'id de l'utilisateur, l'id du film
        user_id = int(float(ligne[0]))
        movie_id = int(float(ligne[1]))

        # Récupérer l'indice de l'utilisateur
        # pour savoir à quelle ligne dans la matrice ajouter ces informations
        indice_user = list_users.index(user_id)

        # Récupérer l'indice du film
        # pour savoir à quelle colonne dans la matrice ajouter ces informations
        indice_movie = list_movies.index(movie_id)

        # Ajouter 1 étant donné que l'utilisateur a vu le film
        matrix_ratings[indice_user][indice_movie] = 1

    # Renvoyer la matrice
    return matrix_ratings
```

Ensuite, nous faisons appel à la méthode créée précédemment afin de récupérer les différentes matrices pour chaque période de temps. Dans la variable "matrix_all", nous stockons les cinq matrices correspondant à chacune des périodes de temps séparément.

In [6]:

```
### Créer les matrices pour chacune des subdivisions de la base de données

nb_division = 5

# Variable qui contiendra les 5 matrices de ratings
# correspondant chacune uniquement à la période concernée
matrix_all = []

# Créer chacune des matrices
for i in range (nb_division):

    filename = "Base de données 20 millions divisée en 5 - Partie " + str(i+1) + ".csv"
    matrice = Matrix_Ratings(tous_les_users, tous_les_films, filename)

    # Ajouter la matrice à la variable reprenant toutes les matrices
    matrix_all.append(matrice)
```

1.6.3 Méthode *Films_Consultes* : Déterminer la liste films vus par un utilisateur

Plus tard, nous aurons besoin de savoir si un utilisateur a déjà vu un film afin de ne pas ajouter un film qu'il a déjà vu à sa liste de recommandations.

Cette méthode prend donc en paramètres :

- La matrice "ratings" où les lignes représentent les utilisateurs, les colonnes représentent les films et le contenu vaut 1 si l'utilisateur a vu le film, 0 sinon.
- La variable "user" correspondant à la position de la ligne liée à l'utilisateur dans la matrice ratings.

La méthode renvoie la liste des films que l'utilisateur a vus : "films_vus", c'est-à-dire tous les films auxquels une valeur de 1 est attribuée pour cet utilisateur dans la matrice "ratings".

In [7]:

```
### Méthode qui permet de trouver l'ensemble des films qu'un utilisateur a vu

def Films_Consultes (ratings, user):

    # Nombre de films dans la base de données
    nb_items = len(ratings[1,:])

    # Liste des films vus par l'utilisateur
    films_vus = []

    # Parcourir tous les films
    for i in range(nb_items):

        # Un utilisateur a vu un film lorsque la valeur de 1 est reprise dans la matrice
        if ratings[user,i] == 1:
            films_vus.append(i)

    # Renvoyer la liste des films vus par l'utilisateur
    return films_vus
```

2 Réalisation de plusieurs algorithmes

Dans cette partie, nous allons coder 7 algorithmes différents nous donnant à chaque fois des recommandations pour chacun des utilisateurs.

2.1 Algorithme de filtrage collaboratif item-based avec similarité cosinus

2.1.1 Réalisation de la matrice de similarité cosinus entre les items

Cette première méthode *Sim_Cosine_Items* permet de réaliser la matrice de similarité entre les items sur base d'une similarité cosinus. Le paramètre nécessaire est la matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon. La méthode renvoie la matrice de similarité entre toutes paires d'items.

In [8]:

```
### Méthode qui fournit une matrice de similarités cosinus entre les items

def Sim_Cosine_Items (ratings):

    # Connaître le nombre d'items présents dans la matrice
    nb_items = len(ratings[1,:])

    print("Réalisation de la matrice de similarités")

    # Matrice transposée (lignes = items, colonnes = users)
    ratings_t = ratings.transpose()

    # Multiplication de la tranposée de la matrice par la matrice elle-même
    mult = ratings_t@ratings

    # Créer la matrice de similarité ne contenant que des zéros pour l'instant
    matrix_sim_cos = numpy.zeros(shape=(nb_items,nb_items))

    # Calcul de chacun des éléments de la matrice
    for i in range(nb_items):
        for j in range (nb_items):

            # Si un des deux films n'a été vu par aucun utilisateur dans cette partie,
            # leur similarité vaut 0
            if mult[i,i]==0 or mult[j,j]==0:
                matrix_sim_cos[i,j] = 0
            else:
                matrix_sim_cos[i,j] = mult[i,j] / ((mult[i,i]**0.5)*(mult[j,j]**0.5))

    # Renvoyer la matrice de similarité
    return matrix_sim_cos
```

2.1.2 Déterminer les *k* plus proches voisins de chaque film

La méthode *Voisins_Items* prend en paramètres la matrice de données "ratings" ainsi que le nombre de voisins *k* à trouver par item. Tout d'abord, nous calculons les similarités entre toutes paires d'items grâce à l'appel de la méthode *Sim_Cosine_Items*.

Pour chaque film, nous récupérons les *k* films qui ont la plus grande similarité avec le film considéré. Cette méthode renvoie alors une liste reprenant les listes des voisins pour chaque film.

In [9]:

```
### Méthode qui renvoie la liste des k plus proches voisins de chaque item

def Voisins_Items (ratings,k):

    # Obtenir la matrice de similarités cosinus entre les items
    matrix_sim_items = Sim_Cosine_Items(ratings)

    print("Calcul des voisins")

    # Liste des voisins pour chaque item
    voisins_items = []

    # Nombre de films
    nb_items = len(matrix_sim_items)

    # Parcourir l'ensemble des items
    for i in range(nb_items):

        # Récupérer les similarités de l'item parcouru
        sim_item = matrix_sim_items[i,:]

        # Passer la similarité du film avec lui-même de 1 à -1
        # car le film ne peut pas être son propre voisin
        sim_item[i] = -1

        # Trouver les voisins de l'item
        voisins_film = sim_item.argsort()[-k:][::-1]

        # Enregistrer les voisins de cet item
        voisins_items.append(voisins_film)

    # Renvoyer la liste des voisins de chaque film
    return voisins_items
```

2.1.3 Les listes de recommandations pour les utilisateurs

Nous avons établi la méthode *Recommandations_ItemBased* qui permet de fournir une liste de films recommandés pour chacun des utilisateurs.

La méthode requiert les paramètres suivants:

- La matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon.
- La variable "nb_recom" qui correspond au nombre de recommandations à proposer pour chaque utilisateur.
- La variable "k" qui correspond au nombre de voisins de chaque film.

Les films recommandés à un utilisateur sont des voisins des films que l'utilisateur a vus. Les films les plus pertinents pour l'utilisateur sont placés au début de la liste. Pour cela, nous reprenons le premier

voisin de chaque film vu par l'utilisateur et l'ajoutons à la liste des films recommandés pour cet utilisateur. Ensuite, nous prenons le deuxième voisin de chaque film vu par l'utilisateur et l'ajoutons à la liste et ainsi de suite jusqu'à ce que la liste de recommandations contienne le nombre d'éléments nécessaires. Evidemment, nous vérifions que les films n'ont pas déjà été vus par l'utilisateur avant de les ajouter à la liste des recommandations.

La méthode renvoie une liste dont les éléments représentent les listes de recommandations pour chaque utilisateur.

In [10]:

```
### Méthode qui fournit une liste de recommandations pour tous les utilisateurs

def Recommandations_ItemBased (ratings, nb_recom, k):

    # Liste des voisins pour chaque film
    voisins_items = Voisins_Items(ratings,k)

    # Nombre d'utilisateurs et nombre de films
    nb_users = len(ratings)
    nb_items = len(ratings[1,:])

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Etablir une liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur
        list_recom_user = []

        # Liste de tous les films vus par l'utilisateur
        films_vu = Films_Consultes(ratings,u)

        # Variable représentant l'indice du film vu par l'utilisateur
        # dont on va regarder les voisins, on regarde d'abord le premier film vu par l'util
        m = 0

        # Variable indiquant le numéro du voisin que l'on regarde
        num_voisin = 0

        # Ajouter des recommandations tant que la liste n'est pas pleine
        # On commence par regarder le premier voisin de tous les films vus par l'utilisateu
        # Puis on regarde le deuxième voisin de tous les films vus par l'utilisateur
        # Et ainsi de suite jusqu'à ce qu'on atteigne le nombre de recommandations nécessai
        while len(list_recom_user) < nb_recom:

            # Film vu par l'utilisateur que l'on consulte
            film_vu = films_vu[m]

            # Ensemble des films voisins du film vu
            voisins = voisins_items[film_vu]

            # On prend le film "num_voisin" voisin du film vu par l'utilisateur
            film_a_evaluer = voisins[num_voisin]

            # Si le film n'est pas encore dans la liste des films recommandés
            # et que l'utilisateur ne l'a pas encore vu, on l'ajoute
            if (film_a_evaluer not in list_recom_user) and (film_a_evaluer not in films_vu)
                list_recom_user.append(film_a_evaluer)

            # Passer au film vu suivant
```

```
            if m < len(films_vu)-1:
                m = m + 1

            # Sauf si on est au dernier film vu,
            # alors on passe au voisin suivant du premier film vu
            else:
                m = 0
                num_voisin = num_voisin + 1

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les recommandations
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste des recommandations pour tous les utilisateurs
    return list_recommandations_all
```

In [125]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins pour chaque film
# Nous décidons de fixer le nombre de voisins pour chaque film
# équivalant au nombre de recommandations pour être certains d'avoir
# suffisamment de voisins pour constituer la liste de recommandations
k = nb_recom

# Créer les recommandations
list_recommandations_itembased = Recommandations_ItemBased(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_itembased)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_itembased[u])
```

```
Réalisation de la matrice de similarités
Calcul des voisins
Liste de recommandations utilisateur 0 :
[586, 265, 186, 176, 751, 139, 179, 1798, 2156, 732, 614, 326, 406, 1490,
517, 331, 889, 298, 312, 174, 391, 165, 735, 465, 663, 1265, 338, 2475, 35
6, 2607, 506, 497, 160, 571, 257, 1581, 1286, 2725, 675, 253, 349, 570, 12
85, 498, 380, 159, 268, 288, 533, 1118, 779, 2540, 211, 252, 382, 591, 31
9, 710, 141, 430, 395, 629, 1605, 573, 2591, 1233, 2432, 823, 666, 883, 11
92, 539, 2295, 812, 196, 712, 191, 311, 337, 158, 1569, 1432, 944, 1274, 1
428, 982, 1090, 2251, 1042, 1208, 1241, 1230, 871, 355, 1395, 2642, 1283,
1451, 2502, 1419]
Liste de recommandations utilisateur 1 :
[586, 751, 1798, 33, 2156, 1221, 1540, 331, 518, 782, 85, 313, 99, 126, 35
6, 577, 292, 131, 486, 497, 2714, 349, 298, 647, 794, 963, 765, 1251, 667,
689, 1369, 1313, 1752, 697, 68, 1708, 36, 32, 1348, 66, 772, 883, 2002, 12
11, 108, 387, 581, 434, 123, 498, 288, 1208, 34, 1539, 553, 1535, 1543, 22
0, 613, 869, 665, 727, 923, 234, 745, 1568, 946, 1503, 1106, 2668, 723, 45
9, 644, 557, 1016, 672, 1038, 1569, 1230, 1944, 2129, 724, 1447, 1672, 229
4, 2034, 1689, 1849, 112, 15, 14, 229, 709, 101, 38, 507, 742, 543, 1068,
```

2.2 Algorithme de filtrage collaboratif user-based avec similarité cosinus

2.2.1 Réalisation de la matrice de similarité cosinus entre les utilisateurs

La méthode *Sim_Cosine_Items* permet de réaliser la matrice de similarité entre les utilisateurs sur base d'une similarité cosinus. Le paramètre nécessaire est la matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon. La méthode renvoie la matrice de similarité entre toutes paires d'utilisateurs.

In [15]:

```
### Méthode qui fournit une matrice de similarités cosinus entre les utilisateurs

def Sim_Cosine_Users (ratings):

    print("Réalisation de la matrice de similarités")

    # Connaître le nombre d'utilisateurs présents dans la matrice
    nb_users = len(ratings)

    # Matrice transposée (lignes = items, colonnes = utilisateurs)
    ratings_t = ratings.transpose()

    # Multiplication de la matrice par sa transposée
    mult = ratings@ratings_t

    # Créer la matrice de similarité ne contenant que des zéros pour l'instant
    matrix_sim_cos = numpy.zeros(shape=(nb_users,nb_users))

    # Calcul de chacun des éléments de la matrice
    for u in range(nb_users):
        for v in range (nb_users):
            matrix_sim_cos[u,v] = mult[u,v] / ((mult[u,u]**0.5)*(mult[v,v]**0.5))

    # Renvoyer la matrice de similarité
    return matrix_sim_cos
```

2.2.2 Déterminer les *k* plus proches voisins de chaque utilisateur

La méthode *Voisins_Users* prend en paramètres la matrice "ratings" ainsi que le nombre de voisins *k* à trouver par utilisateur. Nous avons besoin de la matrice de similarités entre les utilisateurs pour réaliser cette méthode. Celle-ci est obtenue en faisant appel à la méthode "Sim_Cosine_Users".

Pour chaque utilisateur, nous récupérons les *k* utilisateurs qui ont la plus grande similarité avec l'utilisateur considéré. Cette méthode renvoie alors une liste reprenant les listes des voisins pour chaque utilisateur.

In [16]:

```
### Méthode qui renvoie la liste des k plus proches voisins de chaque utilisateur

def Voisins_Users (ratings,k):

    # Obtenir la matrice de similarité entre les utilisateurs
    matrix_sim_users = Sim_Cosine_Users(ratings)

    print("Calcul des voisins")

    # Liste des voisins pour chaque utilisateur
    voisins_users = []

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    # Parcourir l'ensemble des utilisateurs
    for u in range(nb_users):

        # Récupérer les similarités de l'utilisateur parcouru
        sim_user = matrix_sim_users[u,:]

        # Passer la similarité de l'utilisateur avec lui-même de 1 à -1
        # car il ne peut pas être son propre voisin
        sim_user[u] = -1

        # Trouver les voisins de l'utilisateur
        voisins_utilisateur = sim_user.argsort()[-k:][::-1]

        # Enregistrer les voisins de cet utilisateur
        voisins_users.append(voisins_utilisateur)

    # Renvoyer la liste des voisins de chaque utilisateur
    return voisins_users
```

2.2.3 Les listes de recommandations pour les utilisateurs

Nous avons établi la méthode *Recommandations_UserBased* qui permet de fournir une liste de films recommandés pour chacun des utilisateurs sur base du filtrage collaboratif user-based.

La méthode requiert les paramètres suivants:

- La matrice "ratings" où les lignes représentent les utilisateurs et les colonnes représentent les films et dont le contenu vaut 1 si l'utilisateur a vu le film, 0 sinon.
- La variable "nb_recom" représentant le nombre de recommandations à proposer pour chaque utilisateur.
- La variable "k" qui correspond au nombre de voisins de chaque film.

Les films recommandés à un utilisateur sont des films qui ont été vus par les voisins de l'utilisateur. Evidemment, nous vérifions que les films n'ont pas déjà été vus par l'utilisateur avant de les ajouter à la liste des recommandations. Les films les plus pertinents pour l'utilisateur sont placés au début de la liste. Pour cela, nous parcourons les voisins de l'utilisateur un à un et ajoutons tous les films vus par le voisin jusqu'à ce que la liste de recommandations ait atteint le nombre de recommandations nécessaires. La méthode renvoie une liste dont les éléments représentent les listes de recommandations pour chaque utilisateur.

In [17]:

```
### Méthode qui fournit une liste de recommandations pour tous les utilisateurs

def Recommandations_UserBased (ratings, nb_recom, k):

    # Liste des voisins de tous les utilisateurs
    voisins_users = Voisins_Users(ratings, k)

    # Nombre d'utilisateurs et nombre de films
    nb_users = len(ratings)
    nb_items = len(ratings[1,:])

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Etablir une liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur
        list_recom_user = []

        # Liste des films vus par l'utilisateur
        films_vus_u = Films_Consultes(ratings,u)

        # Liste des voisins de l'utilisateur
        voisins = voisins_users[u]

        # Variable permettant d'identifier le voisin dans la liste des voisins,
        # on commence par le 1er voisin
        l = 0

        # Ajouter des recommandations jusqu'à ce que la liste soit pleine
        while len(list_recom_user) < nb_recom:

            # Le voisin à considérer
            v = voisins[l]

            # Liste des films vus par le voisin
            films_vus_v = Films_Consultes(ratings,v)

            # Parcourir tous les films vus par le voisin
            for i in films_vus_v:

                # Ajouter le film à la liste s'il n'y est pas déjà
                # et si l'utilisateur ne l'a pas vu
                # et si on n'a pas atteint le nombre de recomandations à fournir
                if (i not in list_recom_user) and (i not in films_vus_u) and len(list_recom
                    list_recom_user.append(i)

            # Passer au voisin suivant
            l = l + 1

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandations
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [126]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_userbased = Recommandations_UserBased(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_userbased)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_userbased[u])
```

```
Réalisation de la matrice de similarités
Calcul des voisins
Liste de recommandations utilisateur 0 :
[138, 139, 140, 144, 151, 152, 154, 157, 158, 159, 160, 165, 167, 168, 16
9, 171, 173, 174, 177, 178, 179, 181, 186, 187, 189, 190, 191, 194, 208, 2
11, 212, 214, 251, 252, 253, 255, 257, 263, 265, 268, 271, 273, 274, 276,
279, 281, 282, 283, 288, 289, 306, 309, 310, 312, 315, 319, 320, 324, 326,
328, 331, 332, 333, 338, 340, 343, 348, 353, 354, 356, 368, 376, 382, 387,
391, 394, 395, 397, 398, 402, 403, 406, 420, 426, 427, 434, 437, 463, 465,
470, 483, 486, 492, 493, 496, 506, 513, 523, 528, 536]
Liste de recommandations utilisateur 1 :
[14, 15, 24, 29, 31, 32, 33, 34, 35, 36, 38, 39, 43, 50, 53, 54, 56, 57, 6
3, 64, 65, 66, 68, 70, 73, 75, 83, 84, 86, 91, 93, 99, 100, 101, 106, 107,
110, 112, 116, 119, 124, 137, 211, 214, 223, 226, 229, 248, 252, 287, 288,
289, 291, 292, 294, 297, 298, 299, 302, 304, 306, 309, 310, 313, 317, 320,
321, 322, 323, 324, 326, 331, 332, 333, 336, 338, 343, 345, 346, 349, 350,
352, 355, 356, 361, 364, 368, 376, 377, 380, 382, 386, 387, 390, 391, 393,
395, 397, 398, 401]
Liste de recommandations utilisateur 2 :
```

2.3 Algorithme de filtrage collaboratif introduisant une plus grande diversité dans les recommandations

2.3.1 Calculer la diversité d'un ensemble d'éléments

Pour réaliser cet algorithme, nous aurons besoin de pouvoir identifier la diversité présente au sein d'une liste composée de plusieurs items. Nous avons donc réalisé la méthode *Diversity* qui permet de calculer ceci. Elle prend deux paramètres :

- La matrice "matrix_sim_items" regroupant les similarités entre toutes paires d'items de la base de données.
- La liste "items" composée des indices de plusieurs films. On souhaite connaître la diversité de cette liste.

La diversité entre deux items a et b correspond simplement à *1-sim(a,b)* où *sim(a,b)* représente la similarité entre ces deux items. Pour connaître la diversité au sein d'un ensemble d'éléments, il faut alors additionner la diversité entre toutes paires d'items de l'ensemble et diviser cette somme par *((n/2)*(n-1))* où *n* représente le nombre d'éléments repris dans la liste. La méthode renvoie finalement la diversité de la liste calculée.

In [19]:

```
### Méthode qui permet de calculer la diversité entre plusieurs items

def Diversity (matrix_sim_items, items):

    # Nombre d'items au sein de la liste dont il faut étudier la diversité
    nb_items_div = len(items)

    # Variable qui contiendra la somme de 1 - la similarité entre toutes paires d'items
    somme = 0

    # Parcourir toutes paires d'éléments de la liste
    for j in range(0,nb_items_div-1):
        for k in range(j+1,nb_items_div):

            # Récupérer les deux items de la paire
            itemj = items[j]
            itemk = items[k]

            # Ajouter à la somme 1 - la similarité entre la paire d'items
            somme = somme + (1 - matrix_sim_items[itemj,itemk])

    # Calculer la diversité
    div = somme / ((nb_items_div/2)*(nb_items_div-1))

    # Renvoyer la valeur de la diversité entre les éléments de la liste
    return div
```

2.3.2 Trouver l'item apportant la plus grande diversité en l'ajoutant à une liste de recommandations

La méthode *Best_Diverse_Item* permet de trouver l'item qui apporte la plus grande diversité par rapport à d'autres items lorsqu'on l'ajoute à une liste d'items recommandés. Cette méthode prend trois paramètres :

- La liste "recom_actuelle" qui contient tous les items qui sont recommandés.
- La liste "recom_potentielle" qui contient tous les items que l'on peut potentiellement ajouter à la liste des films recommandés.
- La matrice "matrix_sim_items" regroupant la similarité entre toutes paires d'items de la base de données.

Le but de la méthode est alors de sélectionner un seul item parmi tous ceux composant la liste "recom_potentielle", celui qui permet d'avoir la plus grande diversité. Pour cela, nous parcourons tous les éléments de la liste de recommandations potentielles et la diversité est calculée lorsqu'on ajoute cet élément à la liste des recommandations actuelles. Ensuite, l'élément apportant la plus grande diversité est identifié et renvoyé.

In [20]:

```
### Méthode qui calcule les diversités entre une liste de recommandations
### et chacun des éléments, un à un, d'une liste de recommandations potentielles

def Best_Diverse_Item (recom_actuelle, recom_potentielle,matrix_sim_items):

    # Liste qui contiendra la diversité de chacun des items potentiels
    # avec la liste de recommandations actuelle
    diversity_list = []

    # Parcourir tous les éléments de la liste potentielle
    for film_potentiel in recom_potentielle:

        # Liste dont on veut calculer la diversité
        list_for_div = []

        # Y mettre la liste de recommandations déjà constituées
        list_for_div.extend(recom_actuelle)

        # Ainsi que l'élément qu'on pourrait potentiellement ajouter à la liste des recomma
        list_for_div.extend([film_potentiel])

        # Calculer la diversité de la liste
        div = Diversity(matrix_sim_items, list_for_div.copy())

        # Ajouter à la liste des diversités la diversité de ce film avec la liste de recomm
        diversity_list.append(div)

    # Trouver le film avec la plus grande diversité
    indice = ((numpy.array(diversity_list)).argsort()[-1:][::-1])[0]
    film_most_diverse = recom_potentielle[indice]

    # Renvoyer le film avec la plus grande diversité
    return film_most_diverse
```

2.3.3 Etablir des recommandations pour tous les utilisateurs

La méthode *Recommandations_Diversifiees* permet de déterminer des recommandations plus diversifiées pour chacun des utilisateurs.

La méthode requiert les paramètres suivants:

- La matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon.
- La variable "nb_recom" qui correspond nombre de recommandations à proposer pour chaque utilisateur.
- La variable "k" qui correspond au nombre de voisins de chaque film.

Tout d'abord, pour trouver les recommandations pour chaque utilisateur, un ensemble de recommandations de base est calculé pour chaque utilisateur en faisant appel à la méthode *Recommandations_UserBased* qui emploie le système de filtrage collaboratif user-based. Cet ensemble est composé de cinq fois plus de recommandations que ce qui est demandé. Toutefois, nous fixons un plafond de 200 pour le nombre d'éléments que peut contenir cet ensemble. En effet, au-delà de cela, il serait nécessaire à notre algorithme de tourner plus de 1h30, ce qui est beaucoup trop long pour pouvoir fournir des résutlats. Ces recommandations ne correspondent pas aux recommandations finales pour les utilisateurs. Dans l'ensemble des recommandations finales pour un utilisateur, nous ajoutons tout d'abord le premier élément recommandé par l'algorithme de filtrage collaboratif user-based classique. Ensuite, nous ajoutons un à un les éléments permettant d'apporter la plus grande diversité à la liste de recommandations jusqu'à ce que celle-ci contienne le nombre d'éléments demandés. La méthode renvoie une liste dont les éléments représentent les listes de recommandations pour chaque utilisateur.

In [21]:

```
### Méthode pour obtenir les recommandations plus diversifiées pour chaque utilisateur

def Recommandations_Diversifiees (ratings, nb_recom, k):

    # Nombre de recommandations nécessaires multiplié par 5
    nb_recom_mult = nb_recom*5

    # Le nombre de recommandations potentielles est plafonné à 200
    if nb_recom_mult>200:
        nb_recom_mult = 200

    # Liste de recommandations potentielles
    recommandations_base_all = Recommandations_UserBased (ratings, nb_recom_mult, k)

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    print("Calcul des nouvelles recommandations")

    # Obtenir la matrice de similarités cosinus entre les items
    matrix_sim_items = Sim_Cosine_Items(ratings)

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Calcul de la liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur u
        list_recom_user = []

        # Liste de recommandations de base pour l'utilisateur u
```

```
        recommandations_base = recommandations_base_all[u]

        # Ajouter le premier élément de la liste de base
        list_recom_user.append(recommandations_base[0])

        # Enlever cet élément de la liste des films que l'on peut potentiellement ajouter
        del recommandations_base[0]

        # Ajouter des recommandations
        # jusqu'à ce que la liste des recommandations finales soit pleine
        while len(list_recom_user) < nb_recom:

            # Trouver le film apportant le plus de diversité
            # parmi la liste des films potentiels
            film_most_diverse = Best_Diverse_Item(list_recom_user, recommandations_base, ma

            # Ajouter ce film à la liste des recommandations finales
            list_recom_user.append(film_most_diverse)

            # Supprimer ce film de la liste des recommandations potentielles
            indice = recommandations_base.index(film_most_diverse)
            del recommandations_base[indice]

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandation
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [127]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être fournies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_diverses = Recommandations_Diversifiees(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_diverses)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_diverses[u])
```

```
Réalisation de la matrice de similarités
Calcul des voisins
Calcul des nouvelles recommandations
Réalisation de la matrice de similarités
Liste de recommandations utilisateur 0 :
[138, 1468, 1293, 786, 1364, 579, 1072, 996, 864, 1172, 354, 600, 1288, 21
4, 1237, 1218, 711, 892, 212, 1289, 251, 1397, 1554, 1064, 1244, 775, 168,
1555, 761, 1274, 708, 253, 810, 340, 1330, 178, 1430, 658, 606, 1066, 122
2, 171, 622, 426, 348, 1302, 437, 1261, 140, 735, 718, 1338, 1341, 483, 63
2, 1442, 692, 617, 315, 871, 574, 1127, 324, 701, 1508, 1519, 563, 190, 42
0, 398, 513, 208, 1565, 394, 1509, 699, 602, 333, 274, 187, 1451, 434, 104
0, 795, 463, 343, 496, 1363, 536, 271, 634, 1171, 1220, 631, 177, 493, 73
0, 557, 320, 1521]
Liste de recommandations utilisateur 1 :
[14, 791, 350, 749, 468, 297, 562, 214, 649, 294, 336, 642, 630, 638, 627,
648, 119, 619, 457, 640, 743, 633, 620, 508, 302, 621, 408, 474, 635, 580,
476, 39, 643, 652, 137, 655, 650, 467, 443, 520, 570, 31, 484, 626, 24, 32
2, 637, 629, 57, 377, 647, 641, 110, 446, 751, 718, 654, 83, 93, 534, 436,
646, 497, 636, 64, 623, 689, 442, 86, 653, 229, 437, 665, 592, 410, 631, 9
```

2.4 Algorithme de filtrage collaboratif introduisant une plus grande nouveauté au sein des recommandations

2.4.1 Compter le nombre de vues de chaque film

Avant de pouvoir réaliser cet algorithme, il est nécessaire de connaître le degré de nouveauté de chacun des films. Le degré de nouveauté d'un film correspond à la différence entre 1 et le degré de popularité de ce film. Le degré de popularité, quant à lui, représente le nombre d'utilisateurs ayant vu le film divisé par le nombre d'utilisateurs repris dans la base de données. C'est pourquoi nous avons créé la méthode *Vues_Films* qui permet donc de compter, pour chaque film, le nombre d'utilisateurs qui ont vu le film, c'est-à-dire le nombre de 1 dans la ligne correspondant à cet utilisateur dans la matrice "ratings".

In [27]:

```
### Méthode qui compte le nombre de fois que chaque film a été vu

def Vues_Films (ratings):

    # Variable qui contiendra le nombre de fois que chaque film a été vu
    nb_vues_films = []

    # Nombre de films
    nb_items = len(ratings[1,:])

    # Parcourir tous les films
    for i in range(nb_items):

        # Récupérer toute la colonne correspondant au film i
        colonne_film = ratings[:,i]

        # Compter le nombre d'éléments non nuls sur la colonne
        nombre_vues = numpy.count_nonzero(colonne_film)

        # Enregistrer le nombre de vues de ce film
        nb_vues_films.append(nombre_vues)

    # Renvoyer la liste du nombre de vues de chaque film
    return nb_vues_films
```

2.4.2 Calculer le degré de nouveauté de chacun des items

Ensuite, nous avons créé la méthode *Degre_Nouveaute* prenant en paramètre la matrice "ratings" qui est une matrice binaire (le contenu vaut 1 lorsque l'utilisateur a vu le film, 0 sinon) afin de calculer le degré de nouveauté de tous les films de la matrice. Cette méthode renvoie une liste contenant le degré de nouveauté de chacun des films de la base de données.

In [28]:

```
### Méthode qui calcule le degré de nouveauté de tous les films

def Degre_Nouveaute (ratings):

    # Variable qui contient le nombre de vues de chaque film
    vues_film = Vues_Films(ratings)

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    # Rapporter la popularité entre 0 et 1 en fonction du nombre d'utilisateurs
    degre_popu = [x / nb_users for x in vues_film]

    # Calculer le degré de nouveauté de chaque film
    degre_nouveaute = [1-x for x in degre_popu]

    # Renvoyer l'ensemble des degrés de nouveauté
    return degre_nouveaute
```

2.4.3 Etablir des recommandations pour tous les utilisateurs

La méthode *Recommandations_Nouvelles* prend les paramètres suivants :

- La matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon.
- La variable "nb_recom" qui correspond au nombre de recommandations à proposer pour chaque utilisateur.
- La variable "k" qui correspond au nombre de voisins de chaque film.

Pour trouver les recommandations pour chaque utilisateur, un ensemble de recommandations de base est d'abord calculé pour chaque utilisateur en faisant appel à la méthode *Recommandations_UserBased* qui emploie le système de filtrage collaboratif user-based. Cet ensemble est composé de cinq fois plus de recommandations que ce qui est demandé. Ces recommandations ne correspondent pas aux recommandations finales pour les utilisateurs.

Ensuite, nous récupérons la liste reprenant les degrés de nouveauté de tous les films de la base de données en faisant appel à la méthode précédente : *Degre_Nouveaute*.

Après cela, nous constituons une liste de recommandations pour chacun des utilisateurs un à un. Pour ce faire, nous reprenons les degrés de nouveauté des films recommandés par l'algorithme de filtrage collaboratif user-based et ordonnons ces films par ordre de nouveauté décroissante. Les films considérés comme étant les plus nouveaux se retrouvent donc au début de la liste et les moins nouveaux au bout de celle-ci. Alors, il nous suffit simplement de récupérer le nombre de films à recommander parmi les premiers éléments de la liste ordonnée.

Finalement, une liste contenant les listes de recommandations de tous les utilisateurs est constituée et renvoyée par la méthode.

In [29]:

```
### Méthode pour obtenir des recommandations avec plus de nouveauté pour chaque utilisateur

def Recommandations_Nouvelles(ratings, nb_recom, k):

    # Nombre de recommandations nécessaires multiplié par 5
    nb_recom_mult = nb_recom*5

    # Liste de recommandations potentielles pour chaque utilisateur
    recommandations_base_all = Recommandations_UserBased (ratings, nb_recom_mult, k)

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    # Degré de nouveauté de chaque film
    nouveaute_all = Degre_Nouveaute(ratings)

    print("Calcul des nouvelles recommandations")

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Calcul de la liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur
        list_recom_user = []

        # Liste de recommandations de base pour l'utilisateur
        recommandations_base = recommandations_base_all[u]

        # Liste reprenant le degré de nouveauté de chacun des films potentiels
        nouveaute_listrec = [nouveaute_all[i] for i in recommandations_base]

        # Ordonner la liste de recommandations de base par ordre de nouveauté décroissante
        tuple_nouv_indices = sorted(zip(nouveaute_listrec,recommandations_base), reverse=Tr
        recommandations_base_ord = [indice[1] for indice in tuple_nouv_indices]

        # Récupérer les nb_recom recommandations les plus nouvelles
        list_recom_user = recommandations_base_ord[:nb_recom]

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandations
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [128]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_nouvelles = Recommandations_Nouvelles(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_nouvelles)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_nouvelles[u])
```

```
Réalisation de la matrice de similarités
Calcul des voisins
Calcul des nouvelles recommandations
Liste de recommandations utilisateur 0 :
[1453, 2369, 1436, 416, 2476, 2204, 1575, 1385, 1306, 762, 683, 663, 2715,
1512, 1387, 1384, 1383, 1059, 847, 2701, 1635, 1592, 1580, 1493, 1492, 108
2, 717, 2695, 1596, 1554, 1386, 1381, 1046, 758, 738, 445, 431, 2716, 250
2, 1468, 1369, 1177, 1032, 994, 975, 2207, 2006, 1286, 934, 853, 851, 627,
531, 1981, 1912, 1545, 1460, 1433, 1336, 1176, 1169, 732, 547, 409, 354, 2
698, 2444, 2358, 1530, 1339, 731, 565, 2697, 1539, 1072, 887, 775, 605, 43
2, 2696, 2572, 2073, 1178, 1142, 1078, 734, 613, 2663, 1926, 1315, 790, 71
9, 711, 681, 2351, 2326, 1550, 1376, 864, 708]
Liste de recommandations utilisateur 1 :
[1107, 791, 2041, 1462, 620, 127, 1036, 895, 883, 856, 770, 762, 594, 175
3, 1646, 1207, 1162, 1011, 689, 684, 655, 654, 1560, 1223, 1016, 946, 915,
694, 674, 647, 1554, 978, 823, 707, 648, 629, 2211, 1321, 1106, 1058, 893,
826, 1179, 691, 627, 584, 799, 765, 732, 596, 547, 354, 1615, 1448, 1201,
981, 635, 630, 534, 478, 456, 1614, 896, 887, 775, 605, 562, 302, 24, 117
8, 1078, 1026, 870, 818, 734, 613, 608, 217, 711, 233, 1902, 1687, 1115, 1
```

2.5 Algorithme fournissant des recommandations aléatoires

Cette méthode *Recommandations_Aleatoires* permet de recommander aux utilisateurs des films sélectionnés aléatoirement parmi tous les films de la base de données. Pour chaque utilisateur, nous sélectionnons aléatoirement des films parmi les films existants en dehors de ceux déjà vus par l'utilisateur. Des films sont sélectionnés jusqu'à ce que la liste de recommandations soit remplie, c'est-à-dire qu'elle ait atteint le nombre de recommandations demandées pour cet utilisateur. La méthode renvoie ensuite une liste dont les éléments représentent une liste de recommandations pour chaque utilisateur.

In [32]:

```
### Méthode qui détermine des recommandations aléatoires pour chaque utilisateur

def Recommandations_Aleatoires (ratings, nb_recom):

    # Nombre d'utilisateurs et nombre de films
    nb_users = len(ratings)
    nb_items = len(ratings[1,:])

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Parcourir tous les utilisateurs
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur
        list_recom_user = []

        # Liste de tous les films vus par l'utilisateur
        films_vus = Films_Consultes(ratings,u)

        # Ajouter une recommandation à la liste
        # jusqu'à ce qu'on ait atteint le nombre de films nécessaire
        while len(list_recom_user) < nb_recom:

            # Liste parmi laquelle on doit choisir un nombre au hasard
            alea = list(set(range(0,nb_items))-set(list_recom_user)-set(films_vus))

            # Générer un film aléatoirement parmi les films qui peuvent être ajoutés à la l
            film_alea = random.choice(alea)

            # Ajouter le film à la liste des films recommandés à l'utilisateur
            list_recom_user.append(film_alea)

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandations
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [129]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Créer les recommandations
list_recommandations_aleatoires = Recommandations_Aleatoires(matrice, nb_recom)

# Imprimer les recommandations
for u in range(len(list_recommandations_aleatoires)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_aleatoires[u])
```

```
Liste de recommandations utilisateur 0 :
[7227, 9972, 14028, 5516, 7545, 15186, 8212, 672, 12733, 358, 3742, 1493,
9884, 2427, 15559, 9816, 1550, 4524, 12051, 7052, 3165, 2271, 2544, 3187,
5653, 13558, 6720, 9768, 7805, 13736, 2074, 1589, 3322, 9697, 8353, 11050,
7376, 8376, 4602, 5246, 15201, 7450, 15182, 3045, 301, 2710, 1345, 10375,
3853, 10725, 12146, 12699, 9866, 5406, 6887, 15031, 13063, 3927, 3369, 832
4, 5462, 9493, 2016, 1659, 6801, 6932, 3474, 3511, 6133, 8958, 5098, 1193
0, 9961, 12568, 4116, 3653, 9154, 12947, 15059, 7492, 14479, 13410, 9639,
4709, 14163, 12440, 9348, 5602, 4016, 4578, 5711, 3761, 6363, 13964, 5672,
12929, 4522, 512, 13013, 10012]
Liste de recommandations utilisateur 1 :
[9011, 5047, 15471, 1532, 744, 10118, 3150, 7961, 5882, 3993, 12996, 1166,
6100, 14038, 10192, 7962, 8433, 8686, 11687, 4411, 12282, 14461, 14854, 89
28, 5127, 13386, 3685, 14874, 728, 4150, 5641, 7278, 13840, 3299, 610, 127
24, 7268, 2284, 11202, 2164, 12508, 15319, 5881, 5760, 9803, 4153, 9205, 5
054, 4561, 10898, 14939, 10229, 6749, 1949, 8122, 7197, 15065, 12072, 1206
6, 14109, 2934, 485, 11197, 12231, 597, 9880, 4762, 15316, 1844, 7069, 573
2, 7387, 1607, 11294, 13356, 14313, 5282, 2614, 2141, 3609, 4010, 7360, 84
53, 5812, 7500, 4797, 1110, 12811, 3596, 10590, 12361, 8041, 5382, 15513,
```

2.6 Algorithme recommandant les films les plus populaires

La méthode *Recommandations_Populaires* permet de recommander les films les plus populaires pour la période de temps considérée (*t*) aux utilisateurs. Pour cela, nous créons une liste reprenant les films par ordre de popularité, c'est-à-dire que les films les plus populaires sont placés au début de la liste. Plus le nombre d'utilisateurs ayant vu un film est important, plus la popularité de ce film croît également. C'est pourquoi nous faisons appel à la méthode *Vues_Films* créée précédemment et qui permet de connaître le nombre d'utilisateurs ayant vu chaque film.

Pour chaque utilisateur, nous ajoutons le film le plus populaire à la liste des recommandations s'il n'a pas encore été vu par l'utilisateur, ensuite nous ajoutons le deuxième film le plus populaire s'il n'a pas encore été vu par l'utilisateur et ainsi de suite jusqu'à ce que la liste de recommandations ait atteint le nombre de recommandations demandé. Ensuite, la méthode renvoie une liste dont les éléments représentent une liste de recommandations pour chaque utilisateur.

In [81]:

```
### Méthode qui recommande les films les plus populaires aux utilisateurs

def Recommandations_Populaires (ratings, nb_recom, t):

    # Variable qui contient le nombre de fois que chaque film a été vu
    nb_vues_films = numpy.array(Vues_Films(matrix_all[t].copy()))

    # Liste contenant les films par ordre de popularité (1er élément = film le plus populai
    films_populaires = nb_vues_films.argsort()[:][::-1]

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Créer une liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        # Liste de recommandations pour l'utilisateur
        list_recom_user = []

        # Variable qui permet de savoir où on en est dans la liste des films populaires
        i = 0

        # Liste de tous les films vus par l'utilisateur
        films_vus = Films_Consultes(ratings,u)

        # Ajouter des films tant qu'on n'a pas atteint le nombre de recommandations nécessa
        while len(list_recom_user) < nb_recom:

            # Film populaire dont on va regarder si on peut l'ajouter à la liste
            film_popu = films_populaires[i]

            # Recommander un film s'il n'a pas encore été vu par l'utilisateur
            if film_popu not in films_vus:
                list_recom_user.append(film_popu)

            # Passer au film populaire suivant
            i = i+1

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandations
        list_recommandations_all.append(list_recom_user)

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [130]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Créer les recommandations
list_recommandations_populaires = Recommandations_Populaires(matrice, nb_recom, 0)

# Imprimer les recommandations
for u in range(len(list_recommandations_populaires)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_populaires[u])
```

```
Liste de recommandations utilisateur 0 :
[288, 265, 257, 273, 1349, 889, 312, 255, 1843, 169, 281, 310, 174, 331, 1
885, 1031, 289, 252, 397, 139, 2002, 2255, 486, 151, 1208, 268, 338, 211,
332, 406, 1197, 179, 391, 1559, 186, 2097, 1543, 1450, 1609, 571, 152, 34
9, 368, 1760, 2156, 1210, 158, 2173, 588, 2032, 1873, 1520, 1830, 283, 27
1, 160, 138, 1658, 353, 1230, 328, 1241, 2226, 1215, 465, 301, 528, 460, 1
901, 1338, 1804, 1791, 319, 159, 355, 1250, 1193, 382, 1265, 1299, 549, 63
1, 916, 1298, 196, 167, 395, 320, 154, 1644, 1211, 1936, 1302, 1220, 403,
730, 326, 751, 364, 1229]
Liste de recommandations utilisateur 1 :
[288, 1689, 287, 99, 1349, 313, 1843, 1963, 480, 386, 1219, 310, 331, 188
5, 38, 289, 252, 65, 41, 32, 397, 1271, 33, 2002, 66, 2255, 248, 486, 166
4, 63, 1208, 338, 211, 345, 332, 1196, 1197, 391, 1503, 1559, 56, 1327, 44
7, 2097, 1543, 36, 112, 68, 1609, 1465, 35, 1476, 349, 107, 2005, 1886, 31
7, 54, 368, 1760, 344, 2156, 1933, 1210, 2173, 588, 2032, 1873, 73, 1520,
1830, 323, 1337, 433, 247, 52, 108, 1893, 101, 353, 1230, 1668, 1224, 43,
1610, 1241, 2226, 518, 1215, 465, 603, 301, 351, 793, 528, 460, 1209, 190
1, 34, 1338]
Liste de recommandations utilisateur 2 :
```

2.7 Algorithme de filtrage collaboratif avec factorisation matricielle

Cet algorithme consiste à déterminer un certain nombre de facteurs latents pour chaque film et chaque utilisateur. Ces facteurs permettent alors de représenter le film ou l'utilisateur. Pour cela, nous devons passer par un processus itératif permettant d'établir les facteurs latents. Ensuite, sur base de ces facteurs, nous pourrons estimer le fait qu'un utilisateur soit plus enclin à regarder certains films et nous pourrons alors recommander à l'utilisateur les films qu'il regardera le plus probablement.

2.7.1 Détermination des facteurs latents pour chaque film et chaque utilisateur

Nous avons créé la méthode *Facteurs_Latents* permettant de déterminer les facteurs latents caractérisant chaque utilisateur et chaque film. Cette méthode requiert 5 paramètres :

- La matrice "ratings" dont les lignes correspondent aux utilisateurs et les colonnes aux films. Une valeur de 1 est reprise lorsque l'utilisateur a vu le film, 0 sinon. C'est sur base des données reprises dans cette matrice que l'algorithme va pouvoir apprendre et constituer les facteurs latents caractérisant chaque film et chaque utilisateur.
- L'entier positif "nb_fact" correspondant au nombre de facteurs latents qui doivent être déterminés pour chaque film et pour chaque utilisateur.
- L'entier positif "nb_iter" qui représente le nombre d'itérations que nous allons réaliser.
- Le réel positif "learn_rate" qui désigne le taux d'apprentissage, nécessaire dans l'actualisation des facteurs latents à chaque itération.
- Le réel positif "lambdas" qui est également nécessaire dans l'actualisation des facteurs latents à chaque itération.

Initialement, nous attribuons des valeurs aléatoires aux facteurs latents. Ensuite, nous réalisons un processus itératif permettant de faire évoluer les valeurs de ces facteurs latents. A chaque itération, nous parcourons toutes les paires utilisateur-film pour lesquelles une valeur non nulle est inscrite dans la matrice "ratings". Nous estimons alors la valeur prédite par les facteurs latents pour la paire utilisateur-film considérée, selon la formule suivante :

$$prediction = p(u)\ transposée * q(i)$$

Où $p(u)$ correspond à l'ensemble des facteurs latents représentant l'utilisateur u ;

$q(i)$ correspond à l'ensemble des facteurs latents représentant le film i.

Nous vérifions également que la valeur prédite soit bien située entre 0 et 1. Si elle est strictement inférieure à 0, nous lui attribuons une valeur de 0. Et, si elle est strictement supérieure à 1, nous lui accordons une valeur de 1.

Suite à cela, il nous est possible de calculer l'erreur "err" en réalisant la différence entre la valeur réellement rencontrée dans la matrice pour la paire utilisateur u - film i et celle estimée ("prediction").

Ensuite, nous calculons les nouvelles valeurs de chaque facteur latent de chaque film et de chaque utilisateur. Pour ce faire, les formules suivantes sont appliquées :

$$p'(u,f) = p(u,f) + learn\_rate * (err * q(i,f) - lambdas * p(u,f))$$

$$q'(i,f) = q(i,f) + learn\_rate * (err * p(u,f) - lambdas * q(i,f))$$

Où *p(u,f)* représente l'ancienne valeur du facteur latent *f* de l'utilisateur *u* ;

p'(u,f) représente la nouvelle valeur du facteur latent *f* de l'utilisateur *u* ;

q(i,f) représente l'ancienne valeur du facteur latent *f* de l'item *i* ;

q'(i,f) représente la nouvelle valeur du facteur latent *f* de l'item *i*.

Finalement, la méthode renvoie deux matrices : celle regroupant les facteurs latents pour tous les utilisateurs et celles reprenant l'ensemble des facteurs latents pour tous les films.

In [85]:

```
### Méthode qui renvoie les facteurs latents pour chaque utilisateur et chaque film

def Facteurs_Latents (ratings, nb_fact, nb_iter, learn_rate, lambdas):

    ## Initialisation des différentes valeurs ##

    # Nombre d'utilisateurs
    nb_users = len(ratings)

    # Nombre de films
    nb_items = len(ratings[1,:])

    # Matrice de facteurs latents pour les utilisateurs
    # (lignes = utilisateurs, colonnes = facteurs latents)
    fact_users = numpy.zeros(shape=(nb_users,nb_fact))

    # Matrice de facteurs latents pour les films
    # (lignes = films, colonnes = facteurs latents)
    fact_items = numpy.zeros(shape=(nb_items,nb_fact))

    # Générer des nombres aléatoires pour chaque facteur de chaque utilisateur
    for user in range(nb_users):
        for fact in range(nb_fact):

            # Générer un nombre de manière uniforme entre 0,001 et 0,010
            ru = round(random.uniform(0.001, 0.01),10)

            # Générer -1, 0 ou 1
            mu = random.randint(-1, 1)

            # Permettre d'avoir un nombre positif ou négatif
            if mu == 0 :
                fact_users[user][fact] = ru
            else:
                fact_users[user][fact] = ru*mu

    # Générer des nombres aléatoires pour chaque facteur de chaque film
    for item in range(nb_items):
        for fact in range(nb_fact):

            ri = round(random.uniform(0.001, 0.01),10)
            mi = random.randint(-1, 1)
            if mi == 0 :
                fact_items[item][fact] = ri
            else:
                fact_items[item][fact] = ri*mi
```

```
## Processus itératif pour déterminer les facteurs latents ##

# Récupérer les paires utilisateur-film
# où une valeur non nulle est attribuée dans la matrice
users, items = numpy.nonzero(ratings)

# Processus itératif
for iteration in range(nb_iter):
    # Parcourir les paires utilisateur-film
    # pour lesquelles une valeur non nulle est attribuée dans la matrice
    for u,i in zip(users,items):

        # Calculer la prédiction
        prediction = (fact_users[u,:].transpose())@fact_items[i,:]

        # La valeur prédite doit entre comprise entre 0 et 1
        if prediction>1 :
            prediction = 1
        elif prediction<0 :
            prediction = 0

        # Calculer l'erreur
        err = ratings[u][i] - prediction

        # Parcourir les facteurs
        for f in range(nb_fact):

            # Récupérer le facteur de l'utilisateur u
            fact_u = fact_users[u][f]

            # Récupérer le facteur du film i
            fact_i = fact_items[i][f]

            # Mettre à jour la valeur du facteur f de l'utilisateur u
            fact_users[u][f] = fact_u + learn_rate * (err*fact_i - lambdas*fact_u)

            # Mettre à jour la valeur du fateur f de l'item i
            fact_items[i][f] = fact_i + learn_rate * (err*fact_u - lambdas*fact_i)


# Renvoyer les facteurs latents des utilisateurs et ceux des films
return fact_users, fact_items
```

2.7.2 Etablir les recommandations pour les utilisateurs

Une fois que nous disposons des facteurs latents caractérisant chaque utilisateur et chaque film, il nous est possible d'établir les recommandations pour les utilisateurs. Pour ce faire, nous avons créé la méthode *Recommandations_Factorisation*. Cette méthode requiert les paramètres suivantes :

- La matrice "fact_users" contenant les facteurs latents de tous les utilisateurs. Les lignes correspondent aux utilisateurs et les colonnes aux facteurs.
- La matrice "fact_items" contenant les facteurs latents de tous les items. Les lignes correspondent aux items et les colonnes aux facteurs.
- La variable "nb_recom" qui correspond au nombre de recommandations à proposer pour chaque utilisateur.

Tout d'abord, nous parcourons chaque utilisateur afin d'établir les listes de recommandations. Lors de la constitution de la liste de recommandations pour un certain utilisateur, nous parcourons tous les films de la base de données et estimons, sur base des facteurs latents, la valeur prédite dans la matrice pour cette paire utilisateur-film. Après cela, toujours pour le même utilisateur, nous ordonnons les films par ordre de valeur prédite décroissante. Nous ajoutons alors des films de cette liste un à un à la liste de recommandations pour l'utilisateur jusqu'à ce que celle-ci ait atteint le nombre de recommandations demandées. Finalement, la méthode renvoie les listes de recommandations pour tous les utilisateurs.

In [86]:

```
### Méthode qui établit des recommandations sur base de la factorisation matricielle

def Recommandations_Factorisation (fact_users, fact_items, nb_recom):

    # Nombre d'utilisateurs
    nb_users = len(fact_users)

    # Nombre de films
    nb_items = len(fact_items)

    # Liste reprenant les listes de recommandations de chaque utilisateur
    list_recommandations_all = []

    # Créer une liste de recommandations pour chaque utilisateur
    for u in range(nb_users):

        print("Utilisateur "+str(u))

        # Liste de recommandations pour l'utilisateur u
        list_recom_user = []

        # Variable contenant l'estimation de l'évaluation
        # de chaque film de la base de données par l'utilisateur u
        estimations_u = numpy.zeros(shape=(nb_items))

        # Déterminer l'évaluation pour chaque film de la base de données
        for i in range(nb_items):
            estimations_u[i] = (fact_users[u,:].transpose())@fact_items[i,:]
            if estimations_u[i]>1:
                estimations_u[i]=1
            elif estimations_u[i]<0:
                estimations_u[i]=0

        # Liste contenant les films par ordre de valeur estimée décroissante
        # ce sont les films que nous souhaitons recommander
        films_a_rec = estimations_u.argsort()[:][::-1]
        print(films_a_rec)

        # Variable qui permet de savoir où on en est
        # dans la liste des films que nous souhaitons recommander
        element = 0

        # Liste de tous les films vus par l'utilisateur
        films_vus = Films_Consultes(ratings,u)

        # Ajouter des films tant que le nombre de recommandations nécessaire n'est pas atte
        while len(list_recom_user) < nb_recom:

            # Film dont on va regarder si on peut l'ajouter à la liste des recommandations
            film_potentiel = films_a_rec[element]

            # Recommander un film s'il n'a pas encore été vu par l'utilisateur
            if film_potentiel not in films_vus:
                list_recom_user.append(film_potentiel)
```

```
            # Passer au film potentiel à recommander suivant
            element = element+1

        # Ajouter la liste de recommandations pour cet utilisateur
        # à la liste regroupant toutes les listes de recommandations
        list_recommandations_all.append(list_recom_user)

    print()

    # Renvoyer la liste avec les listes de recommandations pour chaque utilisateur
    return list_recommandations_all
```

In [97]:

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Déterminer les facteurs
fact_users, fact_items = Facteurs_Latents (matrice, 15, 200, 0.005, 0.1)

# Créer les recommandations
list_recommandations_factorisation = Recommandations_Factorisation(fact_users, fact_items ,

# Imprimer les recommandations
for u in range(len(list_recommandations_factorisation)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_factorisation[u])
```

```
Utilisateur 0
[ 661  622 1609 ... 9445 4635 6488]
Utilisateur 1
[  661   622  1609 ...  3923  8835 13520]
Utilisateur 2
[  661   622  1609 ... 11229 11230  8368]
Utilisateur 3
[  661   622  1609 ...  9292 11699  8368]
Utilisateur 4
[  661   622  1609 ...  2797 14056  8368]
Utilisateur 5
[  661   622  1609 ... 13310 10903 11733]
Utilisateur 6
[ 661  622 1609 ... 6456 6454 8368]
Utilisateur 7
[ 661  622 1609 ... 6588 6592 8368]
Utilisateur 8
[  661   622  1609 ... 14056  6592  8368]
Utilisateur 9
```

In [102]:

```
### Test

print("Utilisateur "+str(0))
print(fact_users[0,:])
print("Utilisateur "+str(52))
print(fact_users[52,:])
print("Film "+str(21))
print(fact_items[21,:])
print("Film "+str(162))
print(fact_items[162,:])
```

```
Utilisateur 0
[0.27463806 0.24733699 0.22701373 0.22764069 0.27753942 0.24501019
 0.24531082 0.22650905 0.25946275 0.24111785 0.21401244 0.26718806
 0.23517    0.21592873 0.28027589]
Utilisateur 52
[0.27410503 0.24703847 0.22662711 0.22759302 0.27736707 0.2446162
 0.24497843 0.22615172 0.25888946 0.24100461 0.21369362 0.26681671
 0.23481392 0.21557139 0.27989512]
Film 21
[0.27145844 0.24462844 0.2244227  0.22545198 0.27461097 0.24228883
 0.24261946 0.22398363 0.25645432 0.23862882 0.21166607 0.26429824
 0.23249124 0.21361123 0.27716441]
Film 162
[0.27190173 0.24426113 0.22452108 0.22549771 0.27464095 0.2428686
 0.24233367 0.22383133 0.25720575 0.23842728 0.21189693 0.26450354
 0.23280529 0.21403846 0.27767879]
```

3 Métriques évaluant les recommandations

Nous allons étudier deux métriques : la diversité et la nouveauté.

3.1 La diversité

3.1.1 Rassembler le contenu de plusieurs matrices en une seule

Avant de procéder au codage de la métrique de diversité, nous allons nous concentrer sur la construction de la base de données à partir de laquelle les similarités entre les items seront calculées. En effet, afin de calculer la diversité d'un ensemble d'éléments, il est nécessaire de connaître la similarité entre toutes paires d'items composant cet ensemble. Dès lors, nous avons choisi de calculer la similarité cosinus entre les items. Pour ce faire, nous devons disposer d'une matrice reprenant les données précisant quel film a été vu par quel utilisateur. Nous avons choisi que cette matrice correspondrait à la base de données initiale. Pour rappel, cette dernière avait été divisée en 5 parties temporelles et nous avions créé 5 matrices reprenant chacune les données relatives à une période de temps. Ici, la matrice qui sera utilisée pour calculer les similarités correspond au rassemblement des données provenant de toutes les divisions temporelles.

C'est pourquoi nous avons créé la méthode *Rassemblement_Matrices* prenant en paramètre un ensemble de matrices à rassembler et renvoyant la matrice qui reprend les données de l'ensemble des matrices fournies en paramètre.

In [56]:

```
### Méthode qui rassemble les différentes divisions temporelles en une seule matrice

def Rassemblement_Matrices(matrices_divisees):

    # Matrice qui contiendra le rassemblement des différentes divisions
    matrice_complete = matrices_divisees[0]

    # Parcourir chaque matrice de division temporelle
    # sauf la première qui est déjà considérée dans la matrice complète
    for t in range(1,len(matrices_divisees)):

        # Matrice à ajouter à la matrice complète
        matrice_a_ajouter = matrices_divisees[t]

        # Rassembler les deux matrices
        matrice_complete = numpy.maximum(matrice_complete,matrice_a_ajouter)

    # Renvoyer la matrice complète
    return matrice_complete
```

3.1.2 Calcul de la similarité entre les éléments de la matrice complète

Dans cette partie de code, nous appliquons tout d'abord, à notre base de données, la méthode qui vient d'être créée. La variable "matrix_all" avait été créée au point 1.6.2 et reprend les 5 matrices de la division de la base de données initiale. Dès lors, nous rassemblons l'ensemble des données de la base de données initiale en une seule matrice nommée "matrice_init". Cette matrice est donc une matrice binaire où les lignes représentent les utilisateurs et les colonnes les films, une valeur de 1 est inscrite lorsque l'utilisateur a vu le film, 0 sinon.

Ensuite, nous calculons les similarités entre toutes paires d'items. Pour cela, nous faisons appel à la méthode "Sim_Cosine_Items", créée précédemment. Celle-ci nous renvoie une matrice où les lignes et les colonnes représentent les films et le contenu correspond à la similarité entre toutes paires de films.

In [57]:

```
### Application à nos données et calcul de la similarité

# Création de la matrice reprenant les données de la base de données initiale
matrice_init = Rassemblement_Matrices(matrix_all)

# Calcul de la matrice de similarité entre les items sur base de la matrice de données
sim_items_matinit = Sim_Cosine_Items (matrice_init)
```

```
Réalisation de la matrice de similarités
```

3.1.3 Calcul de la diversité moyenne

Nous avons créé la méthode *Diversité_Moyenne* qui permet de calculer la diversité moyenne de plusieurs listes de recommandations. Elle prend comme paramètre "all_recom". Il s'agit d'une liste dont chaque élément représente une liste de recommandations pour un utilisateur. Ce paramètre contient donc les listes de recommandations pour tous les utilisateurs.

Dans cette méthode, nous parcourons chaque liste de recommandations et en calculons la diversité en faisant appel à la méthode "Diversity". Nous avions créé cette méthode lors de la réalisation de l'algorithme de filtrage collaboratif intégrant une plus grande diversité au sein des recommandations. Elle prenait en paramètres la matrice de similarité entre tous les items ainsi que la liste des items composant l'ensemble dont nous souhaitions connaître la diversité. Une fois toutes les diversités connues, il nous est possible de calculer la diversité moyenne qui est renvoyée par la méthode.

In [58]:

```
### Méthode qui permet de calculer la diversité moyenne de plusieurs listes de recommandati

def Diversité_Moyenne (all_recom):

    # Variable qui contiendra la somme des diversités de toutes les listes de recommandatio
    sum_diversity = 0

    # Nombre d'utilisateurs
    nb_users = len(matrice_init)

    # Parcourir tous les utilisateurs
    for u in range(nb_users):

        # Calculer la diversité de la liste de recommandations de l'utilisateur u
        user_diversity = Diversity(sim_items_matinit,all_recom[u])

        # Ajouter la diversité à la somme de toutes les diversités
        sum_diversity += user_diversity

    # Calculer la diversité moyenne
    average_diversity = sum_diversity/nb_users

    # Renvoyer la diversité moyenne
    return average_diversity
```

In [131]:

```
### Test : Application

# Calcul de la diversité

diversity_itembased = Diversité_Moyenne(list_recommandations_itembased)
print("Diversité algo item-based : "+str(diversity_itembased))

diversity_userbased = Diversité_Moyenne(list_recommandations_userbased)
print("Diversité algo user-based : "+str(diversity_userbased))

diversity_divers = Diversité_Moyenne(list_recommandations_diverses)
print("Diversité algo incorporant une plus grande diversité : "+str(diversity_divers))

diversity_nouv = Diversité_Moyenne(list_recommandations_nouvelles)
print("Diversité algo incorporant une plus grande nouveauté : "+str(diversity_nouv))

diversity_alea = Diversité_Moyenne(list_recommandations_aleatoires)
print("Diversité algo aléatoire : "+str(diversity_alea))

diversity_popu = Diversité_Moyenne(list_recommandations_populaires)
print("Diversité algo populaire : "+str(diversity_popu))
```

```
Diversité algo item-based : 0.5955500257079883
Diversité algo user-based : 0.4921866305058579
Diversité algo incorporant une plus grande diversité : 0.6650531415496584
Diversité algo incorporant une plus grande nouveauté : 0.7961232414399302
Diversité algo aléatoire : 0.917623582958598
Diversité algo populaire : 0.29110989625939837
```

3.1.4 Calcul de la diversité globale

Nous souhaitons pouvoir comparer les mesures de diversités obtenues au sein de recommandations établies par les différents algorithmes avec une mesure de diversité globale. De cette manière, nous pourrions identifier si certains algorithmes enferment les utilisateurs dans une bulle ou non. En effet, si la diversité fournie par un des algorithmes est inférieure à la diversité globale, nous pourrons supposer que cet algorithme a tendance à enfermer les utilisateurs dans une bulle. Bien sûr, il sera également nécessaire de faire le même travail avec la nouveauté afin de pouvoir tirer de réelles conclusions.

Dans cette optique, nous estimons que la diversité globale correspond à la diversité déjà présente dans la base de données au temps *t*. Afin de calculer la diversité globale, nous avons créé la méthode *Div_Globale* qui prend un seul paramètre : "t". Ce paramètre représente la période de temps pour laquelle nous souhaitons connaître la diversité globale. Notre méthode utilise la variable "matrix_all". Pour rappel, cette variable contient les 5 matrices représentant chacune une division temporelle de la base de données initiale. Nous récupérons alors la matrice correspondant au temps *t* et nous identifions tous les films qui ont été vus à cette période de temps. Ensuite, nous calculons la diversité présente au sein de tous ces films afin d'obtenir la diversité globale, qui est alors renvoyée par la méthode.

In [112]:

```
# Calculer la diversité de tous les éléments de la base de données
def Div_Globale (t):

    # Récupérer la matrice de données initiale du temps t
    matrice = matrix_all[t]

    # Récupérer les films qui ont été vus dans la matrice de données
    films_vus = list(numpy.nonzero(numpy.asarray(Vues_Films(matrice))))[0]

    # Calculer la diversité de cette liste de films
    return Diversity(sim_items_matinit,films_vus)
```

In [113]:

```
# Application
print(Div_Globale(0))
print(Div_Globale(1))
print(Div_Globale(2))
print(Div_Globale(3))
print(Div_Globale(4))
```

```
0.8007254676675227
0.835238500623237
0.853177023996577
0.8665745199287467
0.9033531591892408
```

3.2 La nouveauté

3.2.1 Calculer la nouveauté d'une liste d'items

Avec la méthode *Novelty*, nous calculons la nouveauté d'un ensemble d'items. Cela nous permettra de connaître la valeur de la métrique de nouveauté pour une liste de recommandations. La valeur de nouveauté d'une liste, telle que nous l'avons définie, correspond à la somme des degrés de nouveauté de tous les items composant cette liste divisée par le nombre d'items composant la liste.

Cette méthode prend deux parmètres :

- La matrice "matrice" qui correspond à une matrice binaire où une valeur de 1 est attribuée lorsque l'utilisateur a vu un film, 0 sinon. Les lignes correspondent aux utilisateurs et les colonnes aux films.
- La liste "items" qui est composée des indices d'un ensemble de films dont on souhaite connaître la nouveauté.

Cette méthode fait tout d'abord appel à la méthode *Degre_Nouveaute* créée précédemment et qui permet de calculer le degré de nouveauté de tous les items présents dans une matrice. Ensuite, nous récupérons uniquement le degré de nouveauté des items contenus dans la liste "items". Nous sommons ces degrés et divisons la somme par le nombre d'items composant la liste afin d'obtenir la mesure de nouveauté de la liste. Finalement, nous renvoyons cette mesure de nouveauté.

In [60]:

```
### Méthode qui calcule la nouveauté d'un ensemble d'éléments

def Novelty (matrice,items):

    # Nombre d'items dans la liste
    nb_items = len(items)

    # Degré de nouveauté de tous les items
    degre_nouv_all = Degre_Nouveaute(matrice)

    # Liste reprenant le degré de nouveauté de chacun des items de la liste
    nouveaute_listrec = [degre_nouv_all[i] for i in items]

    # Somme de tous les degrés de nouveauté des éléments de la liste
    somme_nouveaute = sum(nouveaute_listrec)/nb_items

    # Renvoyer la somme
    return somme_nouveaute
```

3.2.2 Calculer la nouveauté moyenne

Nous avons créé la méthode *Nouveauté_Moyenne* qui permet de calculer la nouveauté moyenne de plusieurs listes de recommandations. Elle prend deux paramètres :

- La variable "t" qui correspond à la période de temps étudiée. Cette variable va permettre de récupérer la matrice correspondant à la division de la base de données initiale pour la période de temps considérée.
- La liste "all_recom" dont chaque élément représente une liste de recommandations pour un utilisateur. Ce paramètre contient donc les listes de recommandations pour tous les utilisateurs.

Dans cette méthode, nous parcourons chaque utilisateur et calculons la nouveauté de la liste de recommandations qui lui a été attribuée. Pour ce faire, nous faisons appel à la méthode *Novelty* que nous venons de créer et qui renvoie la nouveauté d'un ensemble d'items. Nous sommons l'ensemble des nouveautés obtenues pour chaque liste de recommandations et il nous est ensuite possible de calculer la nouveauté moyenne de l'ensemble des listes en divisant la somme par le nombre d'utilisateurs. Finalement, la méthode renvoie la valeur de la nouveauté moyenne.

In [61]:

```
### Méthode qui calcule la nouveauté moyenne de plusieurs listes de recommandations

def Nouveauté_Moyenne (t,all_recom):

    matrice = matrix_all[t]

    # Nombre d'utilisateurs
    nb_users = len(matrice)

    # Variable qui contiendra la somme des nouveautés de toutes les listes de recommandatio
    sum_novelty = 0

    # Parcourir tous les utilisateurs
    for u in range(nb_users):

        # Calculer la nouveauté de la liste de recommandations de l'utilisateur u
        user_novelty = Novelty(matrice,all_recom[u])

        # Ajouter la nouveauté à la somme de toutes les nouveautés
        sum_novelty += user_novelty

    # Calculer la nouveauté moyenne
    average_novelty = sum_novelty/nb_users

    # Renvoyer la nouveauté moyenne
    return average_novelty
```

In [132]:

```
### Test : Application

# Matrice de la période de temps actuelle
t = 0

# Calcul de la nouveauté

novelty_itembased = Nouveauté_Moyenne(t,list_recommandations_itembased)
print("Nouveauté algo item-based : "+str(novelty_itembased))

novelty_userbased = Nouveauté_Moyenne(t,list_recommandations_userbased)
print("Nouveauté algo user-based : "+str(novelty_userbased))

novelty_divers = Nouveauté_Moyenne(t,list_recommandations_diverses)
print("Nouveauté algo incorporant une plus grande diversité : "+str(novelty_divers))

novelty_nouv = Nouveauté_Moyenne(t,list_recommandations_nouvelles)
print("Nouveauté algo incorporant une plus grande nouveauté : "+str(novelty_nouv))

novelty_alea = Nouveauté_Moyenne(t,list_recommandations_aleatoires)
print("Nouveauté algo aléatoire : "+str(novelty_alea))

novelty_popu = Nouveauté_Moyenne(t,list_recommandations_populaires)
print("Nouveauté algo populaire : "+str(novelty_popu))
```

```
Nouveauté algo item-based : 0.7978078859434178
Nouveauté algo user-based : 0.761377946090443
Nouveauté algo incorporant une plus grande diversité : 0.8786198262419243
Nouveauté algo incorporant une plus grande nouveauté : 0.9606741813321454
Nouveauté algo aléatoire : 0.9882336377812436
Nouveauté algo populaire : 0.6393352639786143
```

3.2.3 Calcul de la nouveauté globale

Tout comme pour la diversité, nous avons choisi de réaliser une mesure de nouveauté globale pour chaque période de temps. En comparant celle-ci aux résultats de nouveauté obtenus au sein des recommandations faites par les algorithmes, il nous sera possible d'étudier le potentiel enfermement des utilisateurs dans une bulle.

Afin de calculer la nouveauté globale, nous avons réalisé la méthode *Nov_Globale* prenant le paramètre "t" qui correspond à la période de temps dont nous souhaitons connaître la nouveauté globale. Nous récupérons la matrice, issue de la base de données initiale, correspondant à la période de temps *t*. Nous identifions alors tous les films qui ont été vus à cette période de temps. Ensuite, nous calculons la nouveauté présente au sein de tous ces films afin d'obtenir la nouveauté globale, qui est finalement renvoyée par la méthode.

In [110]:

```
# Calculer la nouveauté de tous les éléments de la base de données
def Nov_Globale (t):

    # Récupérer la matrice de données initiale du temps t
    matrice = matrix_all[t]

    # Récupérer les films qui ont été vus dans la matrice de données
    films_vus = list(numpy.nonzero(numpy.asarray(Vues_Films(matrice))))[0]

    # Calculer la nouveauté de cette liste de films
    return Novelty(matrice,films_vus)
```

In [111]:

```
# test
print(Nov_Globale(0))
print(Nov_Globale(1))
print(Nov_Globale(2))
print(Nov_Globale(3))
print(Nov_Globale(4))
```

```
0.9185940070473472
0.9529339284884254
0.9654251696158553
0.9809357333227847
0.9872150510373219
```

4 Simulation du comportement des utilisateurs

Cinq scénarios différents vont être testés. A chacun de ceux-ci, la matrice, sur laquelle se basent les algorithmes pour fournir les recommandations à la période de temps suivante, sera modifiée en fonction des décisions établies pour le scénario considéré. Une des décisions sera l'intégration ou non de la composante humaine. La seconde considérera le comportement des utilisateurs : ils acceptent toutes les recommandations, ils les acceptent avec une certaine probabilité ou ils n'en acceptent aucune. Les différentes combinaisons seront alors testées formant ainsi nos 5 scénarios.

4.1 Intégrer la composante humaine

Intégrer la composante humaine consiste à considérer les films que les utilisateurs ont vus sans les conseils des systèmes de recommandations. Ces données sont disponibles car elles correspondent à celles reprises dans la base de données initiale. En effet, nous avions divisé celle-ci en 5 matrices, correspondant chacune à une division temporelle, reprises dans la variable "matrix_all". Dès lors, nous allons ajouter à la matrice de données, sur laquelle les algorithmes se basent pour fournir des recommandations, l'ensemble des données reprises dans la matrice de "matrix_all" correspondant à la période de temps suivante. Cela permettra alors d'intégrer la composante humaine.

Pour ce faire, nous avons créé la méthode *Comp_Humaine* qui prend deux paramètres :

- La matrice "matrice_avant" qui correspond à la matrice de données sur laquelle les algorithmes se basent pour établir des recommandations. Il s'agit d'une matrice binaire où une valeur de 1 est attribuée lorsqu'un utilisateur a vu un film, 0 sinon. Les lignes correspondent aux utilisateurs et les colonnes aux films.
- La variable "t" qui représente la période de temps dont il faut intégrer les données initiales à la matrice.

Cette méthode effectue alors le rassemblement de deux matrices : celle "matrice_avant" qui contient les données sur lesquelles les algorithmes se sont basés pour établir des recommandations à la période de temps précédente et la matrice "matrix_all[t]" contenant l'ensemble des données de la base de données initiale pour la période de temps t. Le résultat est stocké dans la variable "matrice_apres" et est renvoyé par la méthode.

In [63]:

```
### Méthode qui permet d'ajouter à la matrice
# les données provenant de la base de données initiale pour la période de temps considérée

def Comp_Humaine (matrice_avant, t):

    # Matrice qui contient le rassemblement des données
    matrice_apres = numpy.maximum(matrice_avant,matrix_all[t])

    # Renvoyer la matrice contenant les deux autres
    return matrice_apres
```

4.2 Simulation du comportement où les utilisateurs acceptent toutes les recommandations

A travers la méthode *Rec_Acceptees*, nous simulons le fait que les utilisateurs acceptent toutes les recommandations qui leur sont suggérées.

Cette méthode prend deux paramètres :

- La matrice "matrice_avant" qui correspond à la matrice de données sur laquelle les algorithmes se basent pour établir des recommandations. Il s'agit d'une matrice binaire où une valeur de 1 est attribuée lorsqu'un utilisateur a vu un film, 0 sinon. Les lignes correspondent aux utilisateurs et les colonnes aux films.
- La liste "all_recom" dont chaque élément représente une liste de recommandations pour un utilisateur. Ce paramètre contient donc les listes de recommandations pour tous les utilisateurs.

Pour simuler l'acceptation de toutes les recommandations, nous constituons la matrice "matrice_apres" qui, initialement, contient l'ensemble des données reprises dans "matrice_avant". Ensuite, nous passons, dans "matrice_apres", une valeur de 1 pour tous les films recommandés auprès de chaque utilisateur. Le fait qu'un utilisateur accepte une recommandation est donc traduit par une valeur de 1 inscrite dans la matrice à l'intersection de cet utilisateur et du film qui lui était recommandé. Etant donné que le principe de cette simulation réside dans le fait que tous les utilisateurs acceptent toutes les recommandations, ce même procédé est réalisé pour toutes les recommandations de l'utilisateur, et ce, pour chaque utilisateur. La méthode renvoie finalement la matrice "matrice_apres" constituée.

In [64]:

```
### Méthode qui simule le fait que les utilisateurs acceptent toutes les recommandations

def Rec_Acceptees (matrice_avant, all_recom):

    # Matrice qui contiendra le rassemblement des données
    matrice_apres = matrice_avant

    # Nombre d'utilisateurs
    nb_users = len(matrice_avant)

    # Parcourir tous les utilisateurs
    for u in range(nb_users):

        # Récupérer la liste de recommandations de l'utilisateur
        recom = all_recom[u]

        # Parcourir la liste de recommandations
        for i in recom:

            # Passer la valeur de 1 dans la matrice car l'utilisateur accepte la recommanda
            matrice_apres[u][i] = 1

    # Renvoyer la matrice reprenant les différentes données
    return matrice_apres
```

In [14]:

```
### Test

# Matrice de données
matrice = numpy.array([[1,1,0,0,1,0,0,1,0,1],[0,0,1,1,0,0,1,0,0,1],[1,1,0,1,0,0,1,0,1,0],[0
#matrice = matrix_all[0]

# Listes de recommandations
all_recom = [[0,3],[2,8],[7,4],[1,5],[0,2]]
#all_recom = list_recommandations_populaires

matric_rec_acceptees = Rec_Acceptees(matrice,all_recom)
print(matric_rec_acceptees)
```

```
[[1 1 0 1 1 0 0 1 0 1]
 [0 0 1 1 0 0 1 0 1 1]
 [1 1 0 1 1 0 1 1 1 0]
 [0 1 1 0 1 1 0 1 0 0]
 [1 1 1 0 1 1 0 0 1 1]]
```

4.3 Simulation du comportement où les utilisateurs acceptent les recommandations avec une certaine probabilité

La méthode *Rec_Acceptees_Proba* simule le fait que les utilisateurs acceptent les recommandations avec une certaine probabilité. Nous avons choisi de fixer cette probabilité à 25 %. Cela signifie qu'une recommandation établie pour un utilisateur a 25 % de chance d'être acceptée par cet utilisateur, c'est-à-dire que ce dernier regarde le film qui lui est recommandé. Les mêmes paramètres que pour la méthode précédente sont introduits. La façon de faire est plutôt équivalente si ce n'est qu'une valeur de 1 n'est pas systématiquement accordée à un film recommandé à un utilisateur. En effet, nous générons un nombre de manière aléatoire entre 0 et 1 et si ce nombre est strictement inférieur à la probabilité choisie (25 %), une valeur de 1 est accordée dans la matrice à l'intersection de l'utilisateur et du film qui lui était recommandé.

Il est important de signaler que si nous effectuons plusieurs simulations sur cette méthode, les résultats seront différents même si les mêmes paramètres sont encodés étant donné qu'une notion d'aléatoire entre en jeu.

La méthode renvoie finalement la matrice élaborée.

In [79]:

```
### Méthode qui simule le fait que
# les utilisateurs acceptent les recommandations avec une certaine probabilité

def Rec_Acceptees_Proba (matrice_avant, all_recom):

    # Probabilité qu'une recommandation soit acceptée
    p = 0.25

    # Matrice qui contiendra le rassemblement des données
    matrice_apres = matrice_avant

    # Nombre d'utilisateurs
    nb_users = len(matrice_avant)

    # Parcourir tous les utilisateurs
    for u in range(nb_users):

        # Récupérer la liste de recommandations de l'utilisateur
        recom = all_recom[u]

        # Parcourir la liste de recommandations
        for i in recom:

            # Passer la valeur de 1 dans la matrice avec une probabilité p
            if random.random() < p:
                matrice_apres[u][i] = 1

    # Renvoyer la matrice reprenant les différentes données
    return matrice_apres
```

In [133]:

```
### Test

# Matrice de données
matrice = numpy.array([[1,1,0,0,1,0,0,1,0,1],[0,0,1,1,0,0,1,0,0,1],[1,1,0,1,0,0,1,0,1,0],[0
#matrice = matrix_all[0]

# Listes de recommandations
all_recom = [[0,3],[2,8],[7,4],[1,5],[0,2]]
#all_recom = list_recommandations_populaires

matric_rec_acceptees = Rec_Acceptees_Proba(matrice,all_recom)
print(matric_rec_acceptees)
```

```
[[1 1 0 1 1 0 0 1 0 1]
 [0 0 1 1 0 0 1 0 0 1]
 [1 1 0 1 0 0 1 0 1 0]
 [0 0 1 0 1 1 0 1 0 0]
 [0 1 0 0 1 1 0 0 1 1]]
```

4.4 Construction des différents scénarios

Grâce aux méthodes créées précédemment, nous pouvons passer à la construction de nos 5 scénarios. Pour cela, nous avons établi la méthode *Simulation_Scenarios* qui prend les paramètres suivants :

- La matrice "matrice_avant" qui correspond à la matrice de données sur laquelle les algorithmes se basent pour établir des recommandations. Il s'agit d'une matrice binaire où une valeur de 1 est attribuée lorsqu'un utilisateur a vu un film, 0 sinon. Les lignes correspondent aux utilisateurs et les colonnes aux films.
- La variable "t" qui représente la période de temps dont il faudra potentiellement intégrer les données initiales à la matrice.
- La liste "all_recom" dont chaque élément représente une liste de recommandations pour un utilisateur. Ce paramètre contient donc les listes de recommandations pour tous les utilisateurs.
- La variable "num_sce" qui indique le numéro de scénario à appliquer.

Nous générons tout d'abord la matrice qui intègre la composante humaine en faisant appel à la méthode *Comp_Humaine* créée précédemment. Ensuite, nous simulons chacun des scénarios en faisant appel à la méthode liée au comportement que l'on souhaite simuler. Les scénarios suivants sont simulés :

- Dans le premier scénario, nous simulons le fait que toutes les recommandations sont acceptées et que la composante humaine est intégrée. Pour cela, nous faisons appel à la méthode *Rec_Acceptees* et passons comme matrice en paramètre celle intégrant la composante humaine.
- Pour le deuxième scénario, la composante humaine est également intégrée et nous supposons que les utilisateurs acceptent les recommandations avec une certaine probabilité. Dès lors, nous faisons appel à la méthode *Rec_Acceptees_Proba* et passons comme matrice en paramètre celle intégrant la composante humaine.
- En ce qui concerne le troisième scénario, la composante humaine est intégrée, mais contrairement aux 4 autres scénarios, nous ne considérons pas la composante algorithmique, c'est-à-dire que nous ne tenons pas compte des recommandations qui ont été établies. Cela revient à simuler le fait que les utilisateurs n'acceptent aucune recommandation. Dès lors, la matrice renvoyée correspond seulement à la matrice intégrant la composante humaine.

- Dans le quatrième scénario, la composante humaine n'est plus considérée et nous supposons que les utilisateurs acceptent toutes les recommandations. Pour cela, nous faisons appel à la méthode *Rec_Acceptees* et passons comme matrice en paramètre "matrice_avant", c'est-à-dire la matrice qui ne contient pas la composante humaine.
- Le dernier scénario n'intègre pas non plus la composante humaine et considère, cette fois, que les utilisateurs acceptent les recommandations avec une certaine probabilité. Dès lors, nous faisons appel à la méthode *Rec_Acceptees_Proba* et passons comme matrice en paramètre "matrice_avant", c'est-à-dire la matrice qui ne contient pas la composante humaine.

Finalement, la matrice constituée selon le scénario choisi est renvoyée.

In [66]:

```
### Méthode qui permet la simulation de chacun des scénarios

def Simulation_Scenarios (matrice_avant, t, all_recom, num_sce):

    # Matrice ajoutant la composante humaine
    matrice_comp_hum = Comp_Humaine (matrice_avant, t)

    # 1er scénario : intégration de la composante humaine
    # & les utilisateurs acceptent toutes les recommandations
    if num_sce == 1:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees (matrice_comp_hum, all_recom)

    # 2ème scénario : intégration de la composante humaine
    # & les utilisateurs acceptent les recommandations avec une probabilité
    if num_sce == 2:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees_Proba (matrice_comp_hum, all_recom)

    # 3ème scénario : intégration de la composante humaine
    # & les utilisateurs n'acceptent aucune recommandation
    if num_sce == 3:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = matrice_comp_hum

    # 4ème scénario : pas de composante humaine
    # & les utilisateurs acceptent toutes les recommandations
    if num_sce == 4:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees (matrice_avant, all_recom)

    # 5ème scénario : pas de composante humaine
    # & les utilisateurs acceptent les recommandations avec une probabilité
    if num_sce == 5:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees_Proba (matrice_avant, all_recom)

    # Renvoyer la matrice modifiée en fonction du scénario
    return matrice_apres
```

In [77]:

```
### Test

# Matrice de données
matrice = matrix_all[0]

# Listes de recommandations
all_recom = list_recommandations_populaires

matrice_suivante = Simulation_Scenarios (matrice, 1, all_recom, 5)
print(matrice_suivante)
```

```
[[1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 [1. 1. 1. ... 0. 0. 0.]
 ...
 [0. 1. 1. ... 0. 0. 0.]
 [0. 1. 1. ... 0. 0. 0.]
 [0. 1. 1. ... 0. 0. 0.]]
```

5 Faire tourner le code et obtenir les résultats

Nous disposons à présent de tous les éléments nécessaires pour faire tourner notre code et obtenir des résultats que nous pourrons ensuite analyser. Dès lors, nous avons créé la méthode *Resultats* permettant d'exécuter le code associé à un certain algorithme pour tous les scénarios et toutes les périodes de temps et de stocker les résultats dans un fichier CSV. La méthode prend alors 3 paramètres :

- Le nombre "nb_recom" correspondant au nombre de recommandations que l'algorithme doit fournir pour chaque utilisateur.
- La variable "nom_algo" reprenant le nom de la méthode permettant d'exécuter l'algorithme que l'on souhaite étudier.
- La variable "k" permettant d'identifier le nombre de voisins à sélectionner pour l'algorithme choisi. Certains algorithmes (celui recommandant des items populaires et celui établissant des recommandations aléatoires) ne requièrent pas ce paramètre. Il sera alors nécessaire de lui attribuer une valeur négative pour qu'il ne soit pas utilisé.

Tout d'abord, nous créons deux matrices : "result_div" et "result_nov", qui contiendront respectivement les valeurs de diversité et de nouveauté obtenues. Les lignes de cette matrice correspondent aux différentes périodes de temps et les colonnes représentent les différents scénarios appliqués.

Nous commençons par réaliser les opérations pour la première de temps étant donné que les résultats obtenus pour celle-ci seront identiques, peu importe le scénario appliqué. Nous faisons alors appel à l'algorithme choisi afin qu'il établisse les recommandations sur base de la matrice correspondant à la première division de la base de données initiale. Nous calculons ensuite la diversité ainsi que la nouveauté présentes au sein des recommandations établies par l'algorithme et nous stockons ces valeurs dans les matrices contenant les résultats. Ensuite, nous parcourons les scénarios un à un et, au sein de ceux-ci, nous parcourons chaque période de temps où nous appelons à nouveau l'algorithme choisi et calculons les mesures de diversité et de nouveauté pour la période de temps et le scénario concernés.

Finalement, afin de pouvoir accéder plus facilement par la suite aux résultats, nous les stockons dans des fichiers CSV. Deux fichiers CSV sont alors créés : un contenant les mesures de diversité et l'autre contenant celles de nouveauté.

Nous avons fait tourner cette méthode 18 fois. En effet, pour chaque algorithme, nous avons testé de recommander 10, 20 et 100 recommandations.

In [87]:

```
### Méthode qui fait tourner le code sur un certain algorithme
# et inscrit les résultats dans un fichier CSV
def Resultats (nb_recom, nom_algo, k):

    # Matrice qui contiendra les résultats de diversité
    # (ligne=période de temps, colonne=scénario)
    result_div = numpy.zeros(shape=(5,5))

    # Matrice qui contiendra les résultats de nouveauté
    # (ligne=période de temps, colonne=scénario)
    result_nov = numpy.zeros(shape=(5,5))

    print("Période de temps 1")

    # Obtenir les recommandations pour la première période de temps
    # Si k<0, ça signifie que ce paramètre n'est pas utile pour cet algorithme
    if k<0 :
        # Si c'est l'algo de recommandations populaires,
        # on doit préciser la période de temps
        if nom_algo.__name__ == "Recommandations_Populaires":

            recom_init = nom_algo(matrix_all[0], nb_recom, 0)
        else:
            recom_init = nom_algo(matrix_all[0], nb_recom)
    else :
        recom_init = nom_algo(matrix_all[0], nb_recom, k)

    # Calcul de la diversité pour cette première période de temps
    # (identique peu importe le scénario)
    result_div[0,:] = Diversité_Moyenne(recom_init)

    # Calcul de la nouveauté pour cette première période de temps
    # (identique peu importe le scénario)
    result_nov[0,:] = Nouveauté_Moyenne(0,recom_init)

    # Parcours des 5 scénarios
    for sce in range(1,6):

        # Réinitialisation des valeurs pour la première période de temps
        matrice = matrix_all[0]
        recom = recom_init

        # Parcours des périodes de temps de 2 à 5 (c'est-à-dire lorsque t vaut de 1 à 4)
        # On ne parcourt pas la première période de temps vu qu'elle a déjà été réalisée
        for t in range(1,5):

            print()
            print("Période de temps "+ str(t+1) + " - Scénario "+ str(sce))

            # Faire évoluer la matrice en fonction du scénario choisi
            matrice = Simulation_Scenarios (matrice.copy(), t, recom.copy(), sce)

            # Etablir les nouvelles recommandations
            if k<0 :
                if nom_algo.__name__ == "Recommandations_Populaires":
```

```
                recom = nom_algo(matrice.copy(), nb_recom, t)
            else:
                recom = nom_algo(matrice.copy(), nb_recom)
        else :
            recom = nom_algo(matrice.copy(), nb_recom, k)

        # Calcul de la diversité
        result_div[t,(sce-1)] = Diversité_Moyenne(recom.copy())

        # Calcul de la nouveauté
        result_nov[t,(sce-1)] = Nouveauté_Moyenne(t,recom.copy())
# Créer le fichier CSV pour la diversité
numpy.savetxt("Algorithme "+ nom_algo.__name__ +" - "+ str(nb_recom) +" recommandations
# Créer le fichier CSV pour la nouveauté
numpy.savetxt("Algorithme "+ nom_algo.__name__ +" - "+ str(nb_recom) +" recommandations
```

In [124]:

```
# Application
nb_recom = 100
k=15
Resultats (nb_recom, Recommandations_Diversifiees, k)
```

```
Période de temps 1
Réalisation de la matrice de similarités
Calcul des voisins
Calcul des nouvelles recommandations
Réalisation de la matrice de similarités

Période de temps 2 - Scénario 1
Réalisation de la matrice de similarités
Calcul des voisins
Calcul des nouvelles recommandations
Réalisation de la matrice de similarités

Période de temps 3 - Scénario 1
Réalisation de la matrice de similarités
Calcul des voisins
Calcul des nouvelles recommandations
Réalisation de la matrice de similarités

Période de temps 4 - Scénario 1
```